Trabajo de Final de Grado

Grado en Ingeniería en Tecnologías Industriales

Optimización del software de correlación digital de imágenes (DIC) en C++

MEMORIA

Autor: Carlos Fernández Ruiz
Director: Miquel Ferrer Ballester
Convocatoria: Septiembre 2019

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Resumen

El objetivo de este trabajo es optimizar un programa de correlación digital de imagen. La idea consiste en hacer una primera fotografía de un objeto sin deformar, por ejemplo, de una probeta de hormigón, y a continuación, hacer una segunda fotografía después de aplicar una carga al objeto. A partir de la imagen de referencia, la imagen actual y una serie de parámetros, el programa busca la deformación que mejor correlacione las dos imágenes.

Este trabajo consiste en la continuación de dos trabajos anteriores: “Development and analysis of a Numpy-based DIC application for strain calculation” por Marc Guillem Zamora Agustí y “Contribució al desenvolupament del programari de correlació digital d’imatges (DIC)” por Antonio Carreras Luque. En el primer trabajo se desarrolla la primera versión del software en *Python*; el segundo proyecto hace un primer intento de optimizar el programa mediante el lenguaje C++. El objetivo de este trabajo es continuar la optimización en C++, y además combinar el trabajo hecho en los dos proyectos para tener una versión del algoritmo completa y optimizada.

Después del proceso de optimización descrito en esta memoria, se consigue disminuir el tiempo de ejecución en un 88,1% respecto el proyecto de Marc Guillem Zamora y un 97,3% respecto el proyecto de Antonio Carreras. Además, también se reduce el consumo de RAM en un 27,4% respecto al original. También se prepara la estructura para una futura implementación de una interfaz de usuario construida en *Python*.

.

Índice

Tablas

Figuras

1. Glosario

Subset	Sección cuadrada alrededor de un punto, donde los desplazamientos y las deformaciones se consideran constantes.
window	Análogo de un *subset*, utilizado en las deformaciones.
ROI	Acrónimo en inglés *Region of Interest*. El área de la imagen de referencia que contiene los puntos que se quieren analizar.
DIC	*(Digital Image Correlation)* Correlación Digital de Imagen
GN	Gauss-Newton
RG	*Reliability Guided,*
IC-GN	Inverse Compositional Gauss-Newton
ZNSSD	Zero-mean Normalized Sum of Squared Difference
(Z)NCC	(Zero-mean) Normalized Cross-Correlation
Scope	En términos del lenguaje de programación de C++: Extensión del código del programa dentro del cual se puede acceder a la variable o declarar o trabajar con ella.
fastcode	Módulo de C++ expuesto a Python, donde se recogen las funciones escritas en C++.
thread	Secuencia de instrucciones que pueden ser ejecutada en paralelo con otras secuencias
multithreading	Entorno donde se ejecutan más de un *thread* simultáneamente.
Race Conditions	Comportamiento impredecible si dos *threads* intentan acceder y editar una misma variable en el mismo instante de tiempo.

2. Prefacio

2.1. Origen del Proyecto

El objetivo global del contexto en el que se desarrolla este proyecto es crear un programa de correlación digital de imagen que encuentre las deformaciones que sufre una pieza sometida a pequeñas deformaciones. Además, este programa debe ser rápido; debe ocupar la mínima memoria posible; debe poder ejecutarse con total independencia sin necesidad de otros programas; y, debe tener una interfaz sencilla y potente para que los usuarios no necesiten un gran conocimiento previo para ejecutarlo. Debido a la dificultad y al volumen de trabajo que implica estos objetivos, el proyecto nace como un trabajo conjunto.

Este proyecto surge como una continuación de dos trabajos anteriores: “Development and analysis of a Numpy-based DIC application for strain calculation” de Marc Guillem Zamora Agustí **[1],** y “Contribució al desenvolupament del programari de correlació digital d’imatges (DIC)” de Antonio Carreras Luque **[2]**. En el primer trabajo, se analiza el algoritmo de correlación digital de imagen y se escribe una implementación en *Python* que encuentra los desplazamientos y las deformaciones. Como continuación, el trabajo de Antonio Carreras tenía como objetivo optimizar el programa traduciendo el código de *Python* a *C++*. Esta traducción a *C++* resulta ser más lenta que la original de *Python*, y existe una necesidad de seguir optimizando el programa.

2.2. Requerimientos Previos

Debido a que el foco de este trabajo consiste en la optimización de un programa escrito en *C++* y la implementación dual de *Python* con *C++*, es recomendable tener unos conocimientos previos de programación. Aún y así, la mayoría de las estrategias de optimización se presentan en formato se pseudocódigo o en diagramas de flujo, por lo que solamente una poca introducción a *C++* y a *Python* será suficiente para seguir los conceptos presentados.

En el caso de estar interesado en seguir la lectura de esta memoria con el código fuente, será necesario descargarse la versión más reciente de *Visual Studio* y seguir los pasos expuestos en el Anexo A.

3. Introducción

La Correlación digital de imagen, abreviado a sus siglas en inglés DIC (*Digital Image Corrolation*), es un método óptico que busca encontrar los desplazamientos y las deformaciones de objetos bajo deformación física a partir de una serie de imágenes. Se usa las propiedades de una textura granular, ya sea natural o artificial, para correlacionar dos imágenes comparando la intensidad de los píxeles en una pequeña área.

En el proyecto de Antonio Carreras Luque se decide traducir el código existente de *Python* a *C++*, debido a que *Python* es un lenguaje interpretado mientras que *C++* es un lenguaje compilado. Los lenguajes compilados están escritos en un lenguaje, y antes de ser ejecutados, se traducen (compilan) a "lenguaje máquina". Por otro lado, los lenguajes interpretados se "interpretan" en vivo en su código fuente. La ventaja de los lenguajes interpretados es que ofrecen más flexibilidad, pero con la consecuencia de ser más lentos. Por lo tanto, la idea del trabajo de Antonio Carreras era que, si se traducía el programa a C++, aumentaría la velocidad de cálculo; pero, la versión en C++ resulto ser más lenta.

3.1. Objetivos del Proyecto

El primer objetivo de este proyecto es optimizar el rendimiento del programa heredado. Se busca recorrer y analizar los cuellos de botella y encontrar diferentes estrategias para reducir su coste computacional. El aumento de rendimiento en este proyecto se define según tres criterios: la reducción del tiempo de cálculo, la disminución del consumo de memoria RAM y la disminución del consumo de recursos del CPU.

Las estrategias de optimización implementadas se pueden resumir en tres fases. La primera consiste en una limpieza general del código. La segunda fase consiste en buscar una librería más rápida a la hora de hacer cálculos matriciales. Finalmente, la tercera fase consiste en paralelizar diferentes tareas para que el ordenador pueda ejecutarlas simultáneamente en vez de secuencialmente, reduciendo así el tiempo de cálculo.

El segundo objetivo de este proyecto es acabar de traducir el programa original de *Python* a C++. Para llevar esto a cabo se combina el uso de los dos lenguajes de programación, aprovechándose de esta manera de las ventajas de cada uno de ellos. Los cálculos más costosos se traducen a C++, manteniendo el resto en *Python*, de esta manera reduciendo el volumen de código a traducir. Además, se aprovecha la flexibilidad de *Python* para dejar preparada una estructura en la que se pueda construir una futura interfaz de usuario potente.

4. Resumen Teórico

En el siguiente apartado se presenta un resumen de la teoría que forma el algoritmo en el que se basa el programa a optimizar. El foco de este trabajo reside en optimizar el código de un programa donde ya están implementados todos los conceptos teóricos resumidos a continuación; por lo que, no se entrará en excesivo detalle sobre cada concepto.

Este capítulo sirve a modo resumen para dar el contexto de los cálculos que forman el código, si se desea entrar en más detalle sobre todos estos conceptos, se recomienda leer el trabajo de Marc Guillem Zamora **[1]** o leer el artículo de NCORR **[3]**, (una versión de una implementación *open source* del algoritmo DIC).

4.1. Transformación del Subset

Se define un *subset* como un conjunto de píxeles centrados sobre un punto (x_c, y_c). En este proyecto, se ha escogido agrupar estos píxeles en cuadrados, y su tamaño es un parámetro que se debe introducir el usuario. En la Fig. 4.1 mostrada a continuación, se representa un ejemplo de un *subset* de 12 píxeles en longitud.

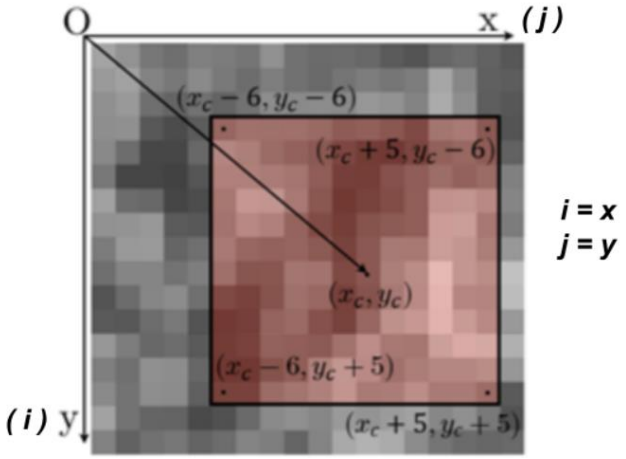

Fig. 4.1 Ejemplo de la configuración de un *subset*. Se muestran los dos tipos de coordenadas: las globales *(x, y)*; y las matriciales *(i, j)*.

A lo largo de este proyecto se usarán dos juegos de coordenadas: las coordenadas globales de imagen, representadas con la notación (x, y), donde la *x* es el eje horizontal y la *y* el eje

vertical; y, las coordenadas matriciales, con notación (i, j), donde la *i* es análoga a la *y*, y la *j* análoga a la *x*. Estos dos sistemas de coordenadas también se representan en la Fig. 4.1. La notación (x, y) se suele utilizar cuando se esté trabajando con la lectura y muestra de imágenes y gráficos, además de la notación de las fórmulas teóricas; por otro lado, la notación (i, j) se utiliza más a nivel de código, una vez las imágenes han sido convertidas a matrices y se tenga que acceder a estas.

La deformación dentro de cada *subset S* se asumen uniforme, por lo que, la elección demasiado grande o pequeña puede llevar a errores de cálculo. Esta deformación viene dada por una transformación lineal de primer orden que se puede expresar con la siguiente relación: **[1]**

Ecuación 4.1

$$\begin{Bmatrix} \tilde{x}_{w_j} \\ \tilde{y}_{w_i} \\ 1 \end{Bmatrix} = O_v + W(\Delta X_{ij};\ P_{tw}) = \begin{Bmatrix} x_{t_c} \\ y_{t_c} \\ 0 \end{Bmatrix} + \begin{bmatrix} 1+\frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & u \\ \frac{\partial v}{\partial x} & 1+\frac{\partial v}{\partial y} & v \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} \Delta x_{t_j} \\ \Delta y_{t_i} \\ 1 \end{Bmatrix}$$

$$\Delta x_{t_j} = x_{t_j} - x_{t_c} \quad ; \quad \Delta y_{t_i} = y_{t_i} - y_{t_c} \quad ; \quad (i,j) \in S$$

En la Ecuación 4.1, ~ se refiere al punto transformado; *u* y *v* son, respectivamente, los desplazamientos horizontales y verticales; el subíndice *c* indica centro; los índices *i, j* corresponden con la posición dentro del *subset* S; y, *t* y *w* se refieren a las imágenes de donde se extrae el valor de la escala de grises. Si *t* ≠ *w*, la transformación ocurre entre la imagen de referencia (*t*) y la imagen actual (*w*); mientras que el caso de *t* = *w* se utiliza para el algoritmo expuesto más adelante en el apartado 4.4. La función $W(\Delta X_{ij};\ P_{tw})$ es la función de deformación (*warping function*), que lleva los puntos desde *t* hasta *w*, según el vector de deformaciones P_{tw}. A continuación, en las ecuaciones *Ecuación 4.2 y Ecuación 4.3,* se expresa el vector P_{tw}, y su análogo en formato matriz W_{tw}:

Ecuación 4.2

$$P_{tw} = \left\{ u, v, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial v}{\partial x}, \frac{\partial v}{\partial y} \right\}^T$$

Ecuación 4.3

$$W_{tw} = \begin{bmatrix} 1+\frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & u \\ \frac{\partial v}{\partial x} & 1+\frac{\partial v}{\partial y} & v \\ 0 & 0 & 1 \end{bmatrix}$$

El objetivo del algoritmo es encontrar, para cada *subset* en la imagen de referencia, un vector de deformaciones optimo ${P_{tw}}^*$ que mejor describa la transformación del *subset* de referencia S_t al *subset* en la imagen transformada (imagen actual) S_w.

4.2. Métricas

El valor óptimo $P_{tw}{}^{*}$ se encontrará cuando la correlación entre el *subset* de referencia S_t y el *subset* deformado S_w sea máxima. Existen dos tipos de coeficientes que miden esta relación entre los *subsets* e indican cuánto de buena es la transformación:

- C_{ZNCC} indica si la correlación es buena cuando se aproxime a 1
- C_{ZNSSD} indica si la correlación es buena cuando se aproxime a 0.

Siendo $G(x_s, y_s) \in [0, 1]$, el valor en escala de grises del punto (x_s, y_s) y $\bar{G}(S)$ el valor medio del *subset* S, que claramente es igual $\sum_{(i,j)\in S} \frac{G(x_j, y_i)}{N}$ (donde N es el número de puntos del *subset*), se presentan a continuación, las expresiones de los dos coeficientes de correlación en la Ecuación 4.4 y en la Ecuación 4.5.

Ecuación 4.4

$$C_{ZNCC} = \frac{\sum_{(i,j)\in S}\left[G(\tilde{x}_{t_j}, \tilde{y}_{t_i}) - \bar{G}(\widetilde{S_t})\right]\left[G(\tilde{x}_{w_j}, \tilde{y}_{w_i}) - \bar{G}(\widetilde{S_w})\right]}{\sqrt{\sum_{(i,j)\in S}\left[G(\tilde{x}_{t_j}, \tilde{y}_{t_i}) - \bar{G}(\widetilde{S_t})\right]^2 \sum_{(i,j)\in S}\left[G(\tilde{x}_{w_j}, \tilde{y}_{w_i}) - \bar{G}(\widetilde{S_w})\right]^2}}$$

Ecuación 4.5

$$C_{ZNSSD} = \sum_{(i,j)\in S}\left[\frac{G(\tilde{x}_{t_j}, \tilde{y}_{t_i}) - \bar{G}(\widetilde{S_t})}{\sqrt{\sum_{(i,j)\in S}\left[G(\tilde{x}_{t_j}, \tilde{y}_{t_i}) - \bar{G}(\widetilde{S_t})\right]^2}} - \frac{G(\tilde{x}_{w_j}, \tilde{y}_{w_i}) - \bar{G}(\widetilde{S_w})}{\sqrt{\sum_{(i,j)\in S}\left[G(\tilde{x}_{w_j}, \tilde{y}_{w_i}) - \bar{G}(\widetilde{S_w})\right]^2}}\right]^2$$

Cabe destacar, que la ventaja de usar estos coeficientes respecto otros criterios es, que los dos no son sensibles al posicionamiento de las imágenes ni a al cambio de intensidad del estado actual. Además, están relacionados según: $C_{ZNSSD} = 2\,(1 - C_{ZNCC})$. Se utiliza uno u otro dependiendo de las circunstancias, por ejemplo: La secuencia de Taylor de C_{ZNSSD} permite encontrar el vector $P_{tw}{}^{*}$ óptimo, mientras que del coeficiente C_{ZNCC} se encontrará la primera correlación.

4.3. Matrices Pre-Calculadas

Para llevar a cabo el algoritmo propuesto en **[1],** es necesario calcular de antemano dos tipos de matrices: las matrices de interpolación y las matrices de derivadas. Los dos siguientes apartados resumen los cálculos de dichas matrices. Para una comprensión más detallada se hace referencia al apartado 3 del trabajo de Marc Guillem Zamora **[1]**. El resumen que se presenta a continuación parte también del trabajo de Antonio Carreras **[2].**

4.3.1. Matrices de Interpolación

Para llevar a cabo el algoritmo propuesto, es necesario realizar una interpolación que de un valor en escala de grises a los puntos entre píxeles. Esta es una interpolación B-spline kernel de grado 5 en dos dimensiones, esta explicada en detalle en el apartado 5.2 de **[1]**.

La interpolación descrita tanto en **[1]** como en **[2]** se puede resumir en la Ecuación 4.6. En esta ecuación se define: $\Delta x = x - \lfloor x \rfloor$, donde $\lfloor x \rfloor$ indica el "suelo" de *x* (redondeando hacia abajo); en otras palabras, Δx indica la parte decimal de *x*. Los superíndices HF y VF indican *Horizontally Flipped i Vertical Flipped* respectivamente; indicando que la fila o columna han girado de manera que el primer valor pasa a ser el último y así sucesivamente.

Ecuación 4.6

$$G(x,y) = \begin{bmatrix} 1 & \Delta y & \Delta y^2 & \Delta y^3 & \Delta y^4 & \Delta y^5 \end{bmatrix} \{[\gamma]^T\}^{HF} [KL]^{shift} [\gamma]^{VF} \begin{bmatrix} 1 \\ \Delta x \\ \Delta x^2 \\ \Delta x^3 \\ \Delta x^4 \\ \Delta x^5 \end{bmatrix}$$

Donde $[\gamma]$ y $[KL]^{shift}$ se expresan en la Ecuación 4.7:

Ecuación 4.7

$$[\gamma] = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & \frac{1}{120} \\ \frac{1}{120} & \frac{1}{24} & \frac{1}{12} & \frac{1}{12} & \frac{1}{24} & -\frac{1}{24} \\ \frac{13}{60} & \frac{5}{12} & \frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} & \frac{1}{12} \\ \frac{11}{20} & 0 & -\frac{1}{2} & 0 & \frac{1}{4} & -\frac{1}{12} \\ \frac{13}{60} & -\frac{5}{12} & \frac{1}{6} & \frac{1}{6} & -\frac{1}{6} & \frac{1}{24} \\ \frac{11}{20} & -\frac{1}{24} & \frac{1}{12} & -\frac{1}{12} & \frac{1}{24} & -\frac{1}{120} \end{bmatrix}$$

$$[KL]^{shift} = \begin{pmatrix} c(-2+\lfloor x \rfloor, -2+\lfloor y \rfloor) & \cdots & c(3+\lfloor x \rfloor, -2+\lfloor y \rfloor) \\ \vdots & \ddots & \vdots \\ c(-2+\lfloor x \rfloor, 3+\lfloor y \rfloor) & \cdots & c(3+\lfloor x \rfloor, 3+\lfloor y \rfloor) \end{pmatrix}$$

Siendo *c* una matriz de coeficientes, obtenida realizando el cálculo unidimensional de la Ecuación 4.8 dos veces: la primera tomando las filas de la matriz; y con nueva matriz tomando cada columna.

Ecuación 4.8

$$c(x) = DFT^{-1}\left\{\frac{DFT\big(G(x)\big)}{DFT\big(\varphi^n(x)\big)}\right\}$$

Si de la Ecuación 4.6 extraemos los productos de $[\Delta y]$ y $[\Delta x]$, se obtiene la siguiente relación:

Ecuación 4.9

$$[Z] = \{[\gamma]^T\}^{HF}\,[KL]^{shift}[\gamma]^{VF}$$

Finalmente, se define las matrices de interpolación en la Ecuación 4.9. La matriz [Z] es una matriz de dimensiones (6x6) que se debe calcular para cada punto de la imagen. Más adelante se explora como el hecho de que esta matriz tenga un tamaño fijo contribuye a uno de los apartados más importantes de la optimización.

4.3.2. Matrices de Derivadas

Además de las matrices de interpolación, es necesario calcular, para cada píxel de la imagen de referencia, dos valores: la derivada en x y la derivada en y:

$$\frac{\partial}{\partial \tilde{x}_{t_j}} G(\tilde{x}_{t_j,}\, \tilde{y}_{t_i}) \;\; y \;\; \frac{\partial}{\partial \tilde{y}_{t_i}} G(\tilde{x}_{t_j,}\, \tilde{y}_{t_i}),$$

Estos valores se pueden calcular a partir de las siguientes ecuaciones:

Ecuación 4.10

$$\frac{\partial}{\partial \tilde{x}_{t_j}} G(\tilde{x}_{t_j,}\, \tilde{y}_{t_i}) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} [Z] \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\frac{\partial}{\partial \tilde{y}_{t_i}} G(\tilde{x}_{t_j,}\, \tilde{y}_{t_i}) = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} [Z] \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Donde [Z] es la matriz de interpolaciones calculada la Ecuación 4.9. Por lo tanto, se recorren todos los píxeles y se construyen dos matrices: una matriz de derivadas en x y la segunda de derivadas en y.

4.4. Algoritmo Gauss-Newton

Este apartado es un resumen y adaptación del apartado 3.3 y 3.3.2 de **[1]**.

El algoritmo de Gauss-Newton se usa para encontrar las raíces de una función, mediante un proceso iterativo. En este caso, encontrar el vector P_{tw} que minimice el coeficiente C_{ZNSSD} para el *subset* de referencia deseado. Para ello, se utiliza la expansión de Taylor de primer orden de C_{ZNSSD} alrededor de ${P_0}^{(n)}$ (Ecuación 4.11). ${P_0}^{(n)}$representa el resultado de la iteración anterior (o la primera correlación en el caso de ser la primera iteración).

Ecuación 4.11

$$C_{ZNSSD}\left({P_0}^{(n)}\right) P^{(n)} \approx C_{ZNSSD}\left({P_0}^{(n)}\right) + \nabla C_{ZNSSD}\left({P_0}^{(n)}\right)^T P^{(n)}$$

Debido a que el objetivo es encontrar el mínimo C_{ZNSSD}, se deriva la Ecuación 4.11 respecto $P^{(n)}$ y se iguala a 0, resultando con la siguiente ecuación:

Ecuación 4.12

$$\nabla\nabla C_{ZNSSD}\left({P_0}^{(n)}\right) P^{(n)} = -\nabla C_{ZNSSD}\left({P_0}^{(n)}\right)$$

El resultado de la Ecuación 4.12, $P^{(n)}$, se convierte en el valor inicial de la siguiente iteración: ${P_0}^{(n+1)} = {P_0}^{(n)} + P^{(n)}$.

Como se explica en el apartado 3.3.2 de **[1]**, para resolver este algoritmo se utiliza el método de la composición inversa de Gauss-Newton, abreviada como: *IG-GN*. La particularidad de este método consiste en dejar que el *subset* de referencia pueda deformar dentro de la imagen de referencia, mientras que ${P_{tw}}^{(n)}$ se mantiene constante en cada iteración y ${P_{tw}}^{(0)}$ se obtiene mediante NCC (mirar apartado 5.1 de **[1]**). Como consecuencia, el $P_{tt} = 0$ es el vector asociado a la deformación óptima. Por lo tanto, se sustituye de la Ecuación 4.12 ${P_0}^{(n)} = 0$ y $P^{(n)} = P_{tt}$ para obtener la Ecuación 4.13.

Ecuación 4.13

$$\nabla\nabla C_{ZNSSD}(0)\ {P_{tt}}^{(n)} = -\nabla C_{ZNSSD}(0)$$

De la Ecuación 4.13, se resuelve el sistema para encontrar el nuevo vector ${P_{tt}}^{(n)}$ mediante la descomposición de Cholesky y el método substitutivo *forward-backward.*

A partir de ${P_{tt}}^{(n)}$, se necesita encontrar la solución óptima en términos de P_{tw} . Para ello, se calcula el vector ${P_{ttw}}^{(n)}$ que se utiliza para encontrar la deformación inicial de la siguiente iteración. Este vector ${P_{ttw}}^{(n)}$ se obtiene de la siguiente ecuación **[1]**:

Ecuación 4.14

$$W_{ttw}^{(n)} \approx W_{tw}^{(n)} \left[W_{tt}^{(n)}\right]^{-1}$$

La Ecuación 4.14 esta expresada en formato matricial; la conversión entre vector y matriz se muestra en las ecuaciones Ecuación 4.2 y Ecuación 4.3. Es importante remarcar, que la aproximación en la Ecuación 4.14 es válida siempre y cuando los componentes (u, v) del vector ${P_{tt}}^{(n)}$ sean suficientemente pequeños.

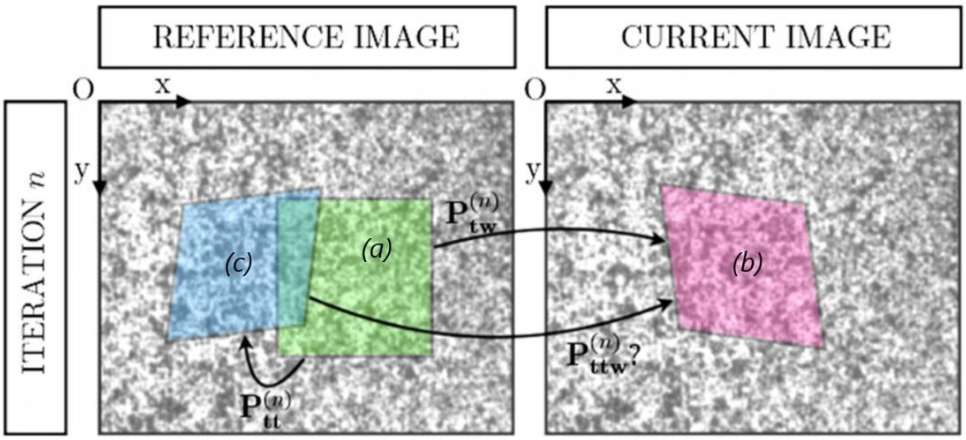

Fig. 4.2 Ejemplo del método IC-GN que muestra el vector $P_{tw}^{(n)}$ deseado. Imagen cortesía de Marc Guillem Zamora **[1]**

Respecto a la Fig. 4.2: primero se calcula el subset deformado en la imagen actual a partir del vector $P_{tw}^{(n)}$ (obtenido como resultado de la iteración anterior o como primera correlación), de donde se saca el subset rosa (b). A continuación, se resuelve la Ecuación 4.13 y se obtiene el vector ${P_{tt}}^{(n)}$ y el subset deformado en la imagen de referencia de color azul (c). A partir de los *subsets* (b) y (c) se encuentra el vector ${P_{ttw}}^{(n)}$. En la siguiente iteración ${P_{ttw}}^{(n)}$ se convierte en el vector ${P_{tw}}^{(n+1)}$: ${P_{tw}}^{(n+1)} = {P_{ttw}}^{(n)}$.

4.4.1. Gradiente y Hessiana

Los cálculos del gradiente y la hessiana de la Ecuación 4.13, se pueden encontrar en el apartado 3.3.3 de **[1]**. La Ecuación 4.15, expresa el gradiente de C_{ZNSSD} respecto de ${P_{tt}}^{(n)}$ en $P_{tt} = 0$.

Ecuación 4.15

$$\nabla C_{ZNSSD}(0) \approx \frac{2}{\sqrt{\sum_{(i,j)\in S}\left[G\left(O_v + \Delta X_{ij}\right) - \bar{G}(S_t)\right]^2}} \cdot \sum_{(i,j)\in S}\left[\left[\frac{G\left(O_v + \Delta X_{ij}\right) - \bar{G}(S_t)}{\sqrt{\sum_{(i,j)\in S}\left[G\left(O_v + \Delta X_{ij}\right) - \bar{G}(S_t)\right]^2}} - \frac{G\left(O_v + W\left(\Delta X_{ij};\, {P_{tw}}^{(n)}\right)\right) - \bar{G}\left(\widetilde{S_w}\right)}{\sqrt{\sum_{(i,j)\in S}\left[G\left(O_v + W\left(\Delta X_{ij};\, {P_{tw}}^{(n)}\right)\right) - \bar{G}\left(\widetilde{S_w}\right)\right]^2}}\right] \cdot \left[\frac{d}{d\left({P_{tt}}^{(n)}\right)}\Bigg|_{{P_{tt}}^{(n)}=0} G\left(O_v + W\left(\Delta X_{ij};\, {P_{tt}}^{(n)}\right)\right)\right]\right]$$

Por otro lado, el cálculo de la hessiana se presenta en la Ecuación 4.16.

Ecuación 4.16

$$\nabla\nabla C_{ZNSSD}(0) \approx \frac{2}{\sum_{(i,j)\in S}\left[G\left(O_v + \Delta X_{ij}\right) - \bar{G}(S_t)\right]^2} \cdot \cdot \sum_{(i,j)\in S}\left[\left[\frac{d}{d\left({P_{tt}}^{(n)}\right)}\Bigg|_{{P_{tt}}^{(n)}=0} G\left(O_v + W\left(\Delta X_{ij};\, {P_{tt}}^{(n)}\right)\right)\right] \cdot \left[\frac{d}{d\left({P_{tt}}^{(n)}\right)}\Bigg|_{{P_{tt}}^{(n)}=0} G\left(O_v + W\left(\Delta X_{ij};\, {P_{tt}}^{(n)}\right)\right)\right]^T\right]$$

En tanto la Ecuación 4.15 como en la Ecuación 4.16, se debe calcular el termino $\frac{d}{d\left({P_{tt}}^{(n)}\right)}\Big|_{{P_{tt}}^{(n)}=0} G\left(O_v + W\left(\Delta X_{ij};\, {P_{tt}}^{(n)}\right)\right)$. Resumiendo el procedimiento, explicado en detalle en **[1]** y **[3]**, este valor se puede extraer a partir de las matrices de derivadas y la distancia del píxel (i, j) al centro del subset. El termino $\frac{d}{d\left({P_{tt}}^{(n)}\right)}\Big|_{{P_{tt}}^{(n)}=0} G(\tilde{x}_{t_j}, \tilde{y}_{t_i})$ es un vector de dimensiones (1x6) que se puede expresar, para cada píxel, como :

Ecuación 4.17

$$\left.\frac{d}{d\left(P_{tt}{}^{(n)}\right)}\right|_{P_{tt}{}^{(n)}=0} G(\tilde{x}_{t_j},\tilde{y}_{t_i}) =$$

$$\left\{\frac{\partial}{\partial\tilde{x}_{t_j}}G(\tilde{x}_{t_j},\tilde{y}_{t_i}),\quad \frac{\partial}{\partial\tilde{y}_{t_i}}G(\tilde{x}_{t_j},\tilde{y}_{t_i}),\quad \frac{\partial}{\partial\tilde{x}_{t_j}}G(\tilde{x}_{t_j},\tilde{y}_{t_i})\,\Delta x_{t_j},\right.$$

$$\left.\frac{\partial}{\partial\tilde{x}_{t_j}}G(\tilde{x}_{t_j},\tilde{y}_{t_i}))\,\Delta y_{t_j}\ \ \frac{\partial}{\partial\tilde{y}_{t_i}}G(\tilde{x}_{t_j},\tilde{y}_{t_i})\,\Delta x_{t_j},\quad \frac{\partial}{\partial\tilde{y}_{t_i}}G(\tilde{x}_{t_j},\tilde{y}_{t_i})\,\Delta y_{t_j}\right\}$$

Finalmente se calcula el termino $G\left(O_v + W\left(\Delta X_{ij};\ P_{tw}{}^{(n)}\right)\right)$ de la Ecuación 4.15 usando las matrices de interpolación de la imagen actual. A partir del vector $P_{tw}{}^{(n)}$, y usando la Ecuación 4.1, se obtienen las nuevas coordenadas del subset en la imagen actual; y a continuación, para cada punto, el valor en escala de grises se interpola siguiendo la Ecuación 4.18:

Ecuación 4.18

$$G(x,y) = [1 \quad \Delta y \quad \Delta y^2 \quad \Delta y^3 \quad \Delta y^4 \quad \Delta y^5]\,[Z]\begin{bmatrix}1\\ \Delta x\\ \Delta x^2\\ \Delta x^3\\ \Delta x^4\\ \Delta x^5\end{bmatrix}$$

$$(\Delta x, \Delta y) = (x - \lfloor x \rfloor, y - \lfloor y \rfloor)$$

4.5. Campo de Desplazamientos

El procedimiento descrito en el apartado 4.4 es válido para cada punto contenido dentro de la región de interés (ROI). Aún y así, calcular cada punto por separado es computacionalmente caro. Para llevar a cabo estas operaciones más eficientemente, se usa el método *Reliability Guided Method (RG)*. **[1].**

Antes de empezar el método RG, se debe encontrar la primera correlación, usando el coeficiente NCC. El cálculo de la primera correlación no se ha modificado en este proyecto y se ha usado el método expuesto en **[2]**. Este punto escogido para iniciar el algoritmo se debe escoger con cuidado en el caso de grandes deformaciones o rotaciones, escogiendo el punto que menos se desplace. El resto de los puntos usarán la información de sus vecinos, debido a que las deformaciones se asumen continuas. Para no propagar los resultados malos (que puedan generar puntos con discontinuidades o con condiciones de luz inapropiadas) se analizan los puntos siguiendo el camino de máximo coeficiente NCC **[2].**

Una vez calculado el vector optimo $P_{tw}{}^{*}$, se calcula el coeficiente NCC para los puntos vecinos, siempre y cuando cumplan dos condiciones: que sea un punto válido y que aún no se haya calculado. Para ser un punto válido, se crea un *subset* centrado en este punto candidato, y todos los puntos de este *subset* tienen que estar dentro de la región de interés (ROI).

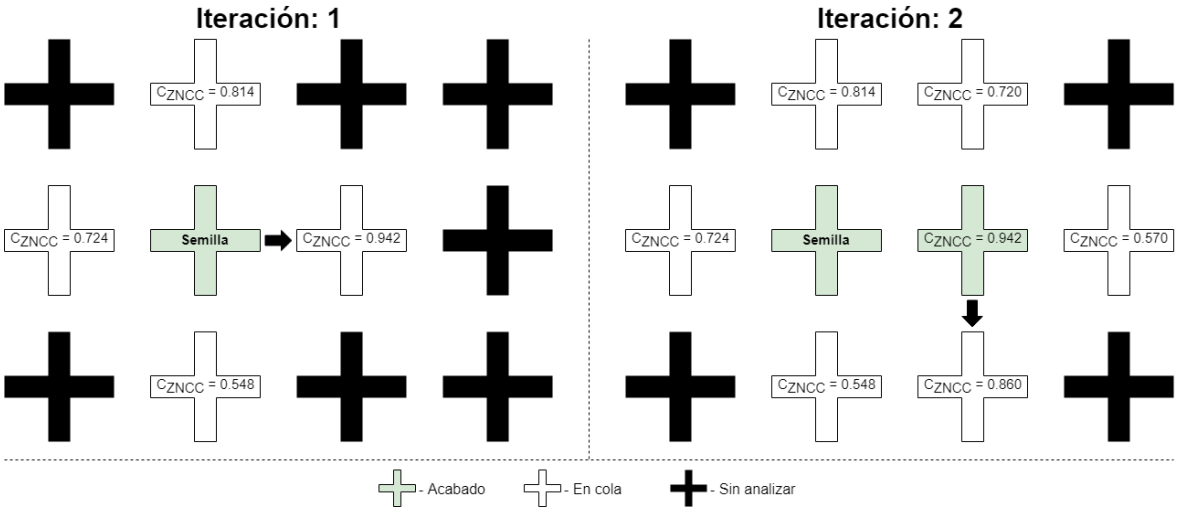

Fig. 4.3 La figura muestra gráficamente la dirección que toma el algoritmo RG-DIC. Cada cruz es el centro de un *subset,* separados entre sí por una distancia dada. El algoritmo sigue el camino de máximo coeficiente NCC. Imagen adaptada de la *Fig.17* de **[3]**

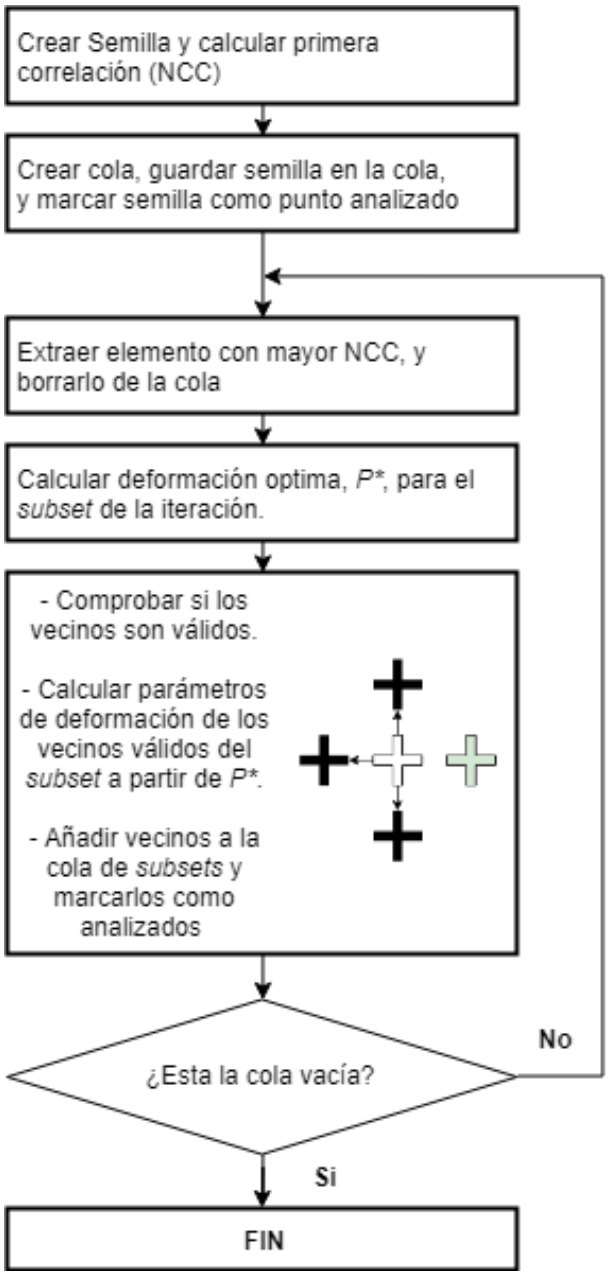

Fig. 4.4 Diagrama de flujo del algoritmo RG-DIC. Las cruces siguen la misma leyenda que la *Fig. 4.3*. Imagen adaptada de la *Fig.18* de **[3]**

Una vez calculados los coeficientes NCC de todos los vecinos, se escoge el vecino que presente mayor coeficiente NCC como siguiente centro a analizar, tal y como se muestra en la Fig. 4.3. El siguiente centro, usará el vector de deformación óptimo de la semilla como su vector de primera correlación. En la figura a continuación, Fig. 4.4, se muestra el diagrama de fujo de la lógica que sigue el algoritmo RG-DIC. Tanto la Fig. 4.3 como la Fig. 4.4 están adaptadas y traducidas del apartado "*dic algorithim*" de la página web de NCORR **[3]**

Una vez se hayan calculado todos los puntos, se debe obtener las deformaciones. Del vector $P_{tw}{}^{*}$, lo único que se usará son los componentes de los desplazamientos horizontales y verticales. Esto es debido que los gradientes de $P_{tw}{}^{*}$ suelen contener mucho ruido, y deben ser filtrados.

Debido a la distancia entre centros de *subset,* para rellenar los puntos que faltan, se debe interpolar los campos de desplazamientos *U* y *V* con una interpolación cubica. **[1]**

4.6. Campo de Deformaciones

En este apartado se presenta un resumen del proceso para obtener el campo de deformaciones seguidos en **[1, 3].**

Antes de todo, se define un análogo de un *subset* llamado *window* que puede tener unas dimensiones diferentes a la longitud del *subset*. Para cada punto del vector de soluciones devuelto por el algoritmo RG-DIC, se construirá una *window*, centrada en el punto, donde se asume que las deformaciones son constantes. Por lo tanto, estos puntos pueden ser proyectados sobre un plano, con una constante y los gradientes como parámetros del plano. Si las coordenadas del centro son (0,0), entonces los dos planos vienen dados por **[1]:**

Ecuación 4.19

$$u_{window}(x,y) = u_{center}^{adjusted} + \frac{\partial u}{\partial x}x + \frac{\partial u}{\partial y}y$$

Ecuación 4.20

$$v_{window}(x,y) = v_{center}^{adjusted} + \frac{\partial v}{\partial x}x + \frac{\partial v}{\partial y}y$$

Formando un sistema de ecuaciones sobre determinado:

Ecuación 4.21

$$\begin{bmatrix} 1 & x_0 - x_c & y_0 - y_c \\ \vdots & \vdots & \vdots \\ 1 & x_n - x_c & y_n - y_c \end{bmatrix} \begin{bmatrix} u_{center}^{adjusted} \\ \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{bmatrix} = \begin{bmatrix} u_0^* \\ \vdots \\ u_n^* \end{bmatrix}$$

Ecuación 4.22

$$\begin{bmatrix} 1 & x_0 - x_c & y_0 - y_c \\ \vdots & \vdots & \vdots \\ 1 & x_n - x_c & y_n - y_c \end{bmatrix} \begin{bmatrix} v_{center}^{adjusted} \\ \frac{\partial v}{\partial x} \\ \frac{\partial v}{\partial y} \end{bmatrix} = \begin{bmatrix} v_0^* \\ \vdots \\ v_n^* \end{bmatrix}$$

En las *Ecuación 4.21* y *Ecuación 4.22*, u_i^* y v_i^* son los desplazamientos obtenidos del vector $P_{tw}{}^*$ de cada punto de la *window*. El subíndice $0 \ldots n$ indica que no todos los puntos de la *window* son válidos; y, que el orden de los puntos no es significante.

Una vez resuelto el sistema de *Ecuación 4.21* y *Ecuación 4.22*, se obtienen los gradientes y se procede a calcular las deformaciones. En este proyecto, se usa el tensor de Green para calcular las deformaciones bidimensionales expresadas en las siguientes tres ecuaciones **[1, 3]:**

Ecuación 4.23

$$E_{xx} = \frac{1}{2}\left[2\frac{\partial u}{\partial x} + \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2\right]$$

Ecuación 4.24

$$E_{yy} = \frac{1}{2}\left[2\frac{\partial v}{\partial y} + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2\right]$$

Ecuación 4.25

$$E_{xy} = \frac{1}{2}\left[\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} + \frac{\partial u}{\partial x}\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\frac{\partial v}{\partial y}\right]$$

Con este proceso se obtienen las deformaciones E_{XX}, E_{YY}, E_{XY} para cada centro analizado, y se debe interpolar nuevamente los valores intermedios para obtener el campo de deformaciones completo. Antes de ello, se puede pasar un filtro adicional para eliminar valores extremos que distorsionen esta interpolación. Estos valores “extraños” se suelen encontrar en las fronteras o discontinuidades. Para filtrar estos valores, se calcula la media y la desviación tipo de cada campo de deformaciones, y se eliminan todos los valores que estén fuera del rango $\mu \pm \sigma \cdot n_{sigm}$; donde n_{sigm} es el parámetro introducido por el usuario, σ la desviación tipo, y μ la media. Este filtro se define con el nombre: filtro sigma.

5. Estructura del Código

Antes de adentrar en el proceso de optimización, se presenta un resumen detallado de la estructura del programa y del código.

Como ya se ha introducido anteriormente, la primera implementación de este programa fue escrita por Marc Guillem Zamora en *Python* **[1]**. Debido a que una gran parte de las funciones que se llamaban desde *Python* estaban escritas "por detrás" en *C++,* se planteó traducir el código a *C++* para mejorar la eficiencia. Esta traducción la empezó Antonio Carreras en **[2]**, aunque supuso dos problemas:

1. **Inacabado**: no se acabó de traducir todo el programa original de *Python*; quedaba por programar los cálculos de las deformaciones y dibujar los gráficos correspondientes.

2. **Rendimiento**: el programa de *C++* era más lento y consumía más memoria RAM que el original de *Python*.

En esta última versión del programa optimizado, se utiliza una combinación entre los lenguajes de programación de *Python* y *C++. Python* es un lenguaje fácil de aprender y utilizar, que tiene muchas funcionalidades y librerías potentes. *C++* es un lenguaje más cercano a "lenguaje máquina", por lo que es más difícil de leer y aprender; pero una vez compilado es más rápido que *Python*.

5.1. Librerías Externas

Además de los programas base, se utilizan librerías externas para ambos lenguajes. A continuación, se muestra una lista con un resumen sobre cada librería. A lo largo de los siguientes apartados se entrará en más detalle sobre el uso de cada librería.

5.1.1. Python:

- ***NumPy*** **[8]** Es una librería científica que ofrece una serie de herramientas para cálculos matriciales. Está escrita en C, esto permite combinar la velocidad de este lenguaje con la simplicidad de *Python*. Se usa para todos los cálculos matriciales necesarios en *Python*.

- ***Matplotlib*** **[9]** se usa en este proyecto principalmente para los gráficos del postproceso y para mostrar al usuario información adicional como las imágenes

seleccionadas o la partición del área de interés.

- ***SciPy* [10] *[9]*** ofrece funcionalidades adicionales a las de *NumPy*. Entre ellas, las herramientas para las interpolaciones cubicas del cálculo de desplazamientos y deformaciones.

- ***Pickle* [11]**. Esta librería permite guardar todos tipo de objetos de *Python* a un fichero binario, que posteriormente se pueden cargar, recuperando estos objetos previamente guardados.

- ***Tkinter* [12].** Esta librería, como las cuatro anteriores, viene instalada por defecto en la instalación estándar de *Python*. Ofrece unas herramientas para construir una interfaz de usuario completa. En este proyecto, se utilizará solo para abrir y seleccionar archivos desde las ventanas de Windows, en vez de tener que escribir la dirección de dichos archivos manualmente.

- ***OpenCv* [13]** es una librería que ofrece herramientas y algoritmos de *computer visión* y de *machine learning*. En la parte de *Python* de este proyecto, se utilizará solamente para cargar las imágenes y convertirlas a matrices de NumPy, sin tener que hacer ninguna operación especial.

5.1.2. C++:

- ***Eigen3* [14]**. Esta librería de *C++* se especializa en cálculos matriciales. Es ligera y sencilla de instalar, ya que está escrita enteramente en ficheros de encabezado (*header files)*. Además, es muy rápida y sus matrices ocupan menos memoria que las de otras librerías; sobre todo con matrices de tamaño fijo que están completamente optimizadas. Gran parte de la optimización de este trabajo se debe a la velocidad que proporcionan sus funciones.

- ***Pybind11* [15].** Esta librería simplifica la comunicación entre *C++* y *Python*. En este proyecto se utilizará para exponer el código de *C++* a *Python* como si fuese una librería adicional.

- ***OpenCv* [13]*Error! Reference source not found.*** Se mantiene el uso de *OpenCv* en C ++ para heredar algunas funciones del trabajo de Antonio Carreras **[2]**, que no se han alterado debido a su bajo impacto en los recursos de CPU y RAM.

5.2. Relación Python – C++

En esta versión del programa, se utilizará *Python* para la interfaz del usuario, para dibujar los gráficos necesarios, y para efectuar algunos cálculos del postproceso. Se reservará el uso del *C++* para resolver los cálculos más "pesados", que necesitan ser muy eficientes. Se mantiene el uso del *Python*, en vez de escribir el programa entero en *C++,* por tres razones:

1. Es más sencillo manipular los datos necesarios para dibujar los gráficos del postproceso. Además, se aprovechan partes del código ya escrito en el trabajo de Marc Guillem Zamora **[1].**

2. Se facilita la construcción de una interfaz de usuario completa, en el caso de la continuación de este proyecto, debido a las librerías potentes dedicadas a ello.

3. La barrera de entrada para los alumnos de la ETSEIB que quieran continuar este proyecto será menor; ya que, estos aprenden *Python* y no *C++* en el grado.

La siguiente figura (Fig. 5.1) expresa de forma visual esta relación entre el usuario, *Python* y *C++.*

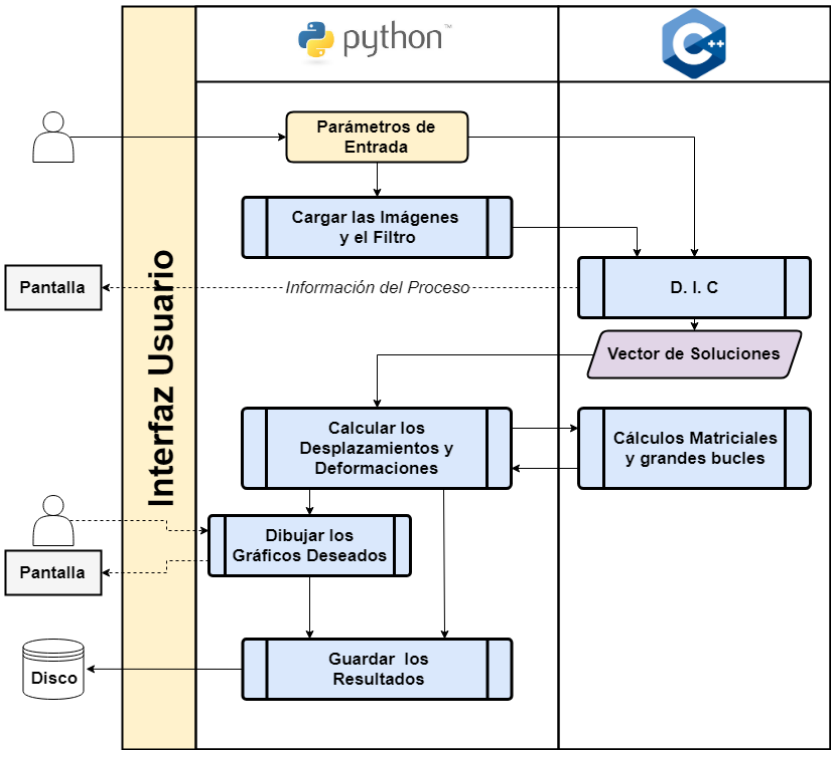

Fig. 5.1: Distribución de las tareas del programa entre *Python* y *C++*

Desde *Python*, programa le pide al usuario que introduzca todos los parámetros de entrada necesarios. Estos se guardan en una clase llamada *inputs*, que facilita el transporte de información a lo largo del programa.

A continuación, se cargan las imágenes deseadas, con la librería *OpenCv*, y se guardan en el formato de matriz de *NumPy*. Después de que el usuario este satisfecho con los datos que ha introducido, se procede a llamar a la función DIC que está compiladas en *C++*.

La comunicación entre los objetos y métodos de *Python* y *C++* se hace mediante una librería llamada *pybind11*. Esta librería simplifica el proceso de comunicación entre los dos lenguajes, y está configurada en este proyecto para que exponga a *Python* las funciones de *C++*. Desde *Python* se debe importar el código de *C++* como si fuese una librería adicional, en este caso llamada *fastcode*, y llamar a las funciones de la forma estándar de *Python*:

```
import fastcode # Nombre del Proyecto de C++ expuesto a Python
vector_soluciones_dic = fastcode.DIC( *kargs )
```

Por el lado de *C++,* se debe definir un fichero de encabezado donde se recojan las funciones que se van a exponer a *Python*. Este fichero, *Binding.h,* se debe incluir al final del *main.cpp*, para que queden todas las funciones definidas antes de ser expuestas. En el Anexo A se explica con más detalle el proceso de exponer las funciones de *C++* a *Python.* La función DIC, del módulo *fastcode*, devuelve un vector de soluciones con los vectores de deformaciones óptimos, $P_{tw}{}^{*}$, de cada centro analizado. Este vector es un vector de tuplas, donde cada tupla contiene el centro del *subset* analizado (x_c, y_c), el vector $P_{tw}{}^{*}$ y el orden en el que ha sido analizado. Ej.: si es el primer centro analizado, *cuenta = 1*; si es el segundo *cuenta = 2*; y así para todos los centros.

Ecuación 5.1

$$dic\_solutions = \left\{\left(\{ x_c, y_c \}, \left\{ u, v, \frac{du}{dx}, \frac{du}{dy}, \frac{dv}{dx}, \frac{dv}{dy} \right\}, cuenta \right), (\dots), \dots \right\}$$

Este vector de tuplas de *C++* se transforma en una lista de tuplas de *Python*. Con esta lista de soluciones, se construyen las dos matrices de desplazamientos U y V, y se interpolan los valores intermedios con la función *interpolate.griddata* de *SciPy* **[10]**. A partir de estas matrices, se calculan las matrices de deformaciones y se vuelve a interpolar los valores intermedios. Parte del cálculo de estas matrices de deformación se hace en *C++*. Esto es debido a que se necesita recorrer un bucle grande, y sale a cuenta traducirlo a *C++* ya que es más eficiente con los cálculos iterativos.

Finalmente, con todas las matrices construidas, se dibujan los gráficos pertinentes. Además, a lo largo del programa se usa la librería *Pickle* **[11]** para ir guardando los objetos de *Python* en un fichero binario de extensión *dicfile*.

5.3. *Module.cpp* del proyecto *fastcode* en C++

Desde *Python* se pasan una serie de parámetros necesarios a la función DIC de *C++* (dentro del módulo *fastcode*). Estos parámetros se guardan dentro de una estructura de *C++* también llamada *inputs*. La ventaja de guardarlos en una estructura es que se facilita, a nivel de código, el paso de estos a través de las funciones. Además, si se quiere añadir o borrar un parámetro, solo se tiene que cambiar en la definición de esta clase, en el archivo de encabezado *RG_GN.h,* facilitando a la escalabilidad del código. La siguiente tabla, *Tabla 5.1*, resume los parámetros de entrada de la función DIC:

Tabla 5.1 Parámetros de entrada de la función DIC

Parámetro	Descripción
Reference	Imagen de Referencia, se introduce como matriz *Eigen3* porque *pybind11* solo soporta conversión entre matrices *NumPy* y *Eigen3*.
Current	Imagen deformada. Ídem que *Reference.*
FiltrROI	Array de *Eigen3* que indica con un "1" los píxeles que se encuentran dentro de la región de interés, y con un "0" los píxeles que se encuentran fuera.
lado_subset	Dimensión del subset, en píxeles.
cnt_space	Distancia entre los centros de cada subset.
search_window	Ventana de búsqueda de la primera correlación.
error	Error máximo admisible en el algoritmo DIC.
cutoff	Número máximo de iteraciones.
seed_list	Lista de las semillas introducidas.
skip_cutoff	Opción de saltarse los centros que no han encontrado solución dentro de las iteraciones permitidas.
show_progres_bar	Mostrar las barras de progreso en pantalla.
verbose	Mostrar información adicional en pantalla.

Antes de entrar en el algoritmo RG_GN en sí, se debe hacer unos cálculos previos. Estos cálculos se encuentran en el fichero *PreComputations.cpp*; y las definiciones de las funciones y clases, en el archivo de encabezado del mismo nombre: *PreComputations.h*.

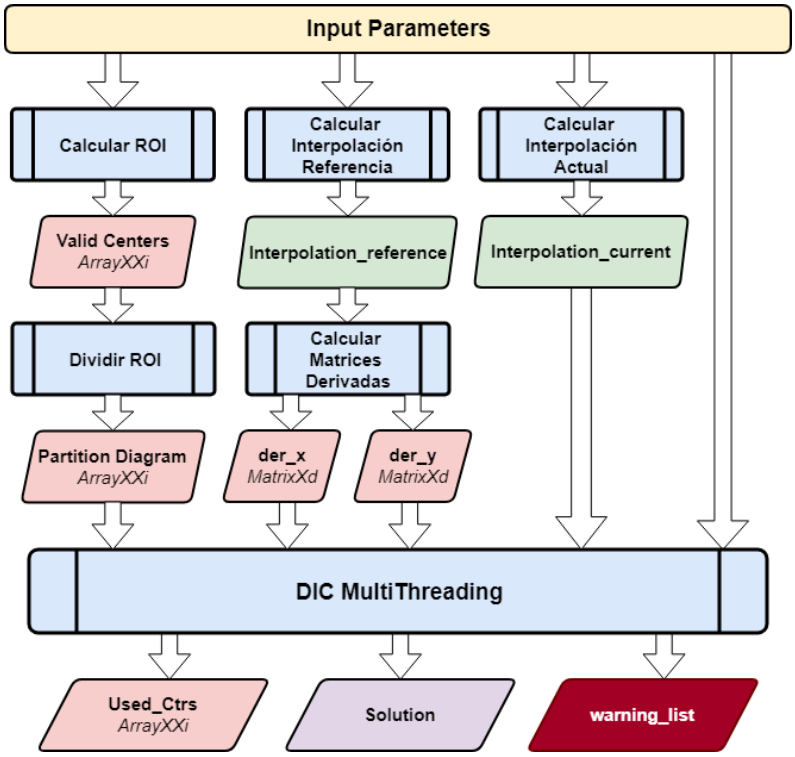

Fig. 5.2 Diagrama de flujo de la función DIC. Las cajas azules representan funciones que se llaman dentro de la función DIC; las cajas rojas son matrices de *Eigen3*; y las cajas verdes son clases propias definidas en el código.

1. **Matriz de Centros Válidos (*valid_ctrs*)** A partir de la imagen de la región interés introducida, se calcula la matriz de centros válidos. Esta matriz expresa con unos la zona de la imagen de referencia donde se puede encontrar los centros de los *subsets* y con ceros donde no. Está construida con un objeto *ArrayXXi* de *Eigen3*. El formato *Array* es un formato especial de representar las matrices que facilita el acceso y las operaciones por coeficientes **[14]**; el subíndice *i* indica que son matrices con valores enteros.

2. **Diagrama de Partición:** Una vez calculada la matriz de centros válidos, se calcula la partición de esta según el número de semillas introducidas. Ej.: si se pasan tres semillas, se dividirá la matriz en tres partes de áreas aproximadamente iguales. La metodología para la partición se explica en más detalle en el apartado 6.3.2. Se puede ver un ejemplo de su estructura en la Fig. 5.3

3. **Matrices de derivadas *(der_x y der_y).*** A partir de las matrices de interpolación de la imagen de referencia, se calculan las matrices de derivadas. Una vez estas matrices son calculadas se borran las matrices de interpolación de la imagen de referencia.

4. **Matrices de Interpolación de la imagen actual (*interpolation_current*).** Las matrices de interpolación se guardan dentro de una clase (*Interpol_Mats*) que internamente se guardan como un vector de matrices. Esta clase se construye para facilitar el acceso a este vector y proporcionar un método seguro de acceso que no lance un error si se intenta acceder a un elemento fuera del rango de dicho vector. (En el Anexo B, se detalla la estructura de esta clase).

A continuación, se inicializan las tres variables de salida:

- ***Used_Ctrs*:** Una matriz de las mismas dimensiones que la imagen de referencia. Con un "1" se indica que centros de *subsets* han sido analizados, y con un "0" cuales aún no. Inicialmente, se marcan todas las posiciones con un "0", excepto las posiciones de las semillas, que se marcan como ya analizadas.

- ***Solution.*** Tal y como se explica en el apartado anterior, es un vector de tuplas que contiene las soluciones de los centros analizados correctamente (Ecuación 5.1)

- ***Warning_list.*** En el programa se ha protegido contra dos posibles errores que causan que se cierre inesperadamente. Para no perder la información de los centros que han encontrado problemas, se recoge en un vector los centros en cuestión con su mensaje de error asociado:

  Ecuación 5.2

  $$warning\_list = \{ (\{x_c, y_c\}, "Mensaje\ de\ Error"), (\dots), \dots \}$$

  En este proyecto se recogen dos tipos de errores. Cuando se encuentra uno de estos dos errores, se salta al siguiente centro en la cola de *subsets* y así no se propaga el error a los vecinos.

  1. "*Cutoff limit reached*". Este mensaje indica que el centro en cuestión ha superado el límite de iteraciones.

  2. *"Out of Current_Image Bounds"*. El algoritmo intenta hacer una deformación al *subset* de referencia demasiado elevada, y esta se encuentra fuera del rango de las imágenes.

Estos tres objetos se inicializan vacíos y se pasan, junto a las otras variables comentadas, como referencias a la función *DIC_Multithreading*. Es importante remarcar que, si a las funciones de *C++* no se pasan las variables como referencia o puntero, el compilador copiara todas las variables de entrada, duplicando el espacio consumido en la memoria RAM.

5.4. Multithreading

La función *DIC_Multithreading* se encarga de preparar el programa para que se pueda ejecutar el algoritmo RG_GN en paralelo utilizando los diferentes *cores* de la máquina.

En los ordenadores modernos, se pueden ejecutar varios programas, y varias funciones dentro de estos programas, en paralelo. Esto es debido a que internamente la máquina tiene diferentes unidades de procesamiento que pueden trabajar independientemente unas de las otras. Se define un *thread*, como una secuencia de instrucciones que puede ser ejecutada en paralelo con otras secuencias en un entorno de *multithreading* **[6]**. Se aprovecha esta funcionalidad para reducir el tiempo de cálculo del algoritmo, ejecutando partes de este simultáneamente.

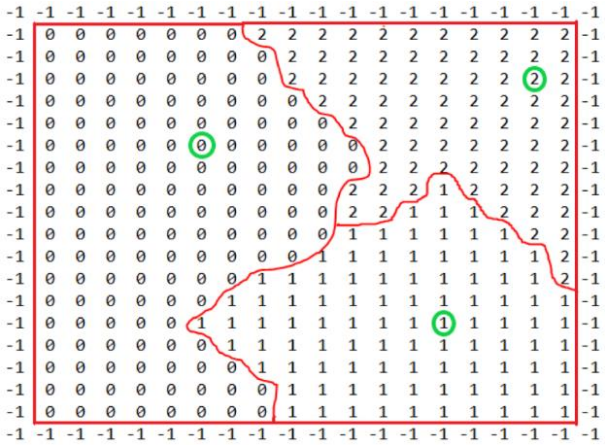

Fig. 5.3 Ejemplo de un diagrama de partición. La región de interés está dividida en 3 zonas, marcadas con los índices: *0, 1* y *2*. El *-1* indica fuera de la ROI. Los círculos verdes muestran las semillas iniciales de cada subregión.

Para llevar a cabo esta implementación, primero se ha de dividir el programa en diferentes tareas independientes. En el caso del algoritmo DIC, la mejor manera de paralelizar, es dividiendo el área de la región de interés en diferentes partes, y que cada *thread* del programa ejecute, por separado, cada una de estas subregiones. **[3][4].**

Esta división se hace mediante el diagrama de partición. Este diagrama consiste en una matriz, de las dimensiones de la imagen de referencia, que marca con un número entero positivo la región que le corresponde a cada *thread*. Esta partición se explica en más detalle en la sección 6.3-*Multithreading*; pero, se muestra la Fig. 5.3 para visualizar la estructura de esta partición.

La primera parte de la función consiste en la preparación de cada *thread,* iniciando un bucle que depende del parámetro de entrada $N_{threads}$. Como cada *thread* ejecuta el algoritmo RG-GN por separado, se necesita crear unas variables propias que se guardan en una clase llamada *thread_data* (mirar Anexo B). En esta clase se guardan variables como: la semilla, la correlación inicial para dicha semilla, la matriz de la subregión de interés y la matriz de centros

usados por ese *thread*. Además, cada clase de *thread_data* tiene su propio vector de soluciones y de *warnings*.

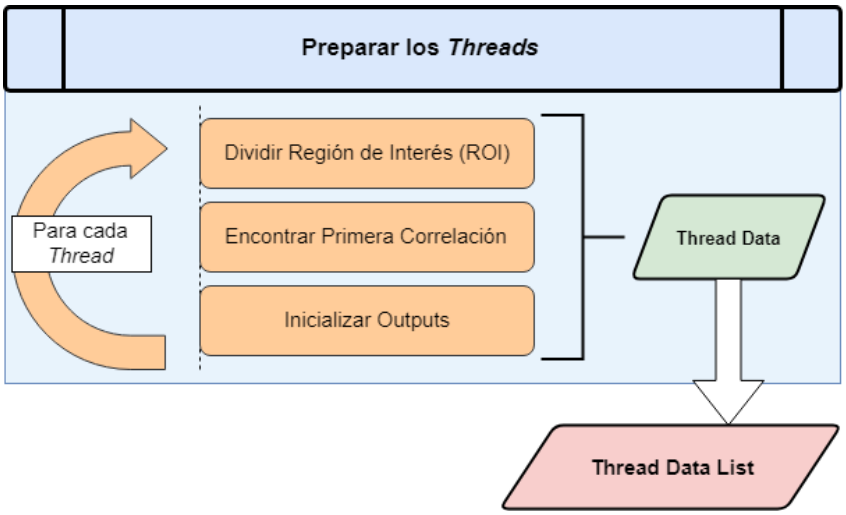

Fig. 5.4 Preparación de los *threads*, inicialización de sus clases *thread_data* e inicialización de la lista de clases *thread_data_list*.

Por portabilidad, se guardan todas las instancias de *thread_data* en un vector. La posición dentro de este vector $(0, 1, \dots, N_{threads} - 1)$ corresponde con el identificador del *thread*, que, a su vez, indica la subregión que le corresponde dentro del *partition_diagram*.

A partir de *thread_data_list*, se llama a la función *launch_threads()* que se encarga de ejecutar todos los *threads* dentro de la lista. A partir de este punto, el programa estará ejecutando $N_{threads} + 1$ procesos paralelos: el número de *threads* indicado en $N_{threads}$, más el *thread* central (*main thread)* desde donde se llaman al resto. Este *main thread* se ocupa de consultar el estado de todos los otros procesos y verificar cuando estos han terminado.

A cada *thread* se le pasa: una instancia de la clase *thread_data* que pueden escribir y leer con independencia; objetos que comparten con el resto de *threads,* de solo lectura; y objetos que comparten con el resto de *threads,* de escritura y lectura.

Dos *threads* pueden consultar simultáneamente una variable compartida si los dos solamente la leen o si uno la lee y otro la escribe. El problema surge, si dos *threads* intentan escribir sobre una variable al mismo tiempo; este comportamiento es impredecible y se denomina *Race Conditions* **[6]**. Para evitar este último caso, se utiliza el objeto *mutex*, que se implementa para impedir que dos *threads* accedan a una misma variable en el mismo instante.

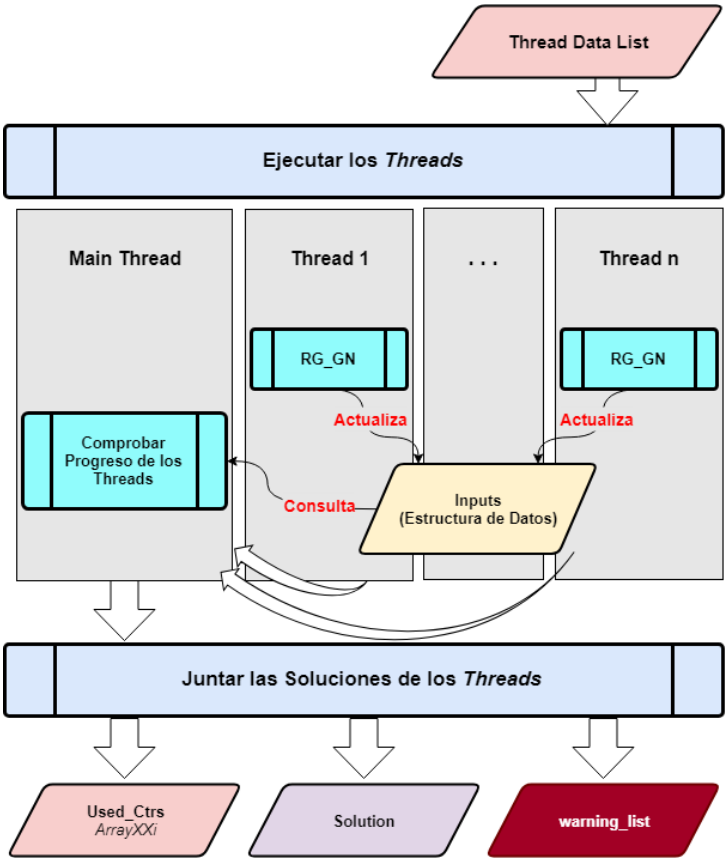

Fig. 5.5 Ejecución del algoritmo RG_GN en cada *thread*. El *main thread* comprueba, mediante la clase *inputs*, el progreso de los otros *threads*. Una vez se termine el análisis en todos los *threads*, el *main thread* se encarga de cerrarlos y de juntar toda la información en los tres objetos: *Used_Ctrs*, *Solution* y *warning_list*

A medida que los *threads* van resolviendo *subsets*, se actualiza la variable *n_centers* de la clase *inputs*. El *main thread* consulta esta variable, para imprimir por pantalla el progreso del algoritmo. Para no entrar en *Race Conditions*, y que dos *threads* intenten editar la clase *inputs* a la vez, se utiliza un *mutex* llamado *inputskey*, que bloquea la clase *inputs*, obligando a que solo un *thread* pueda escribir en ella a la vez.

Al acabar el algoritmo RG_GN, cada *thread* actualiza la variable *thread_finished* de su instancia de la clase *thread_data*. Entonces, el *main thread* consulta esta variable en todos los *threads*, y una vez todos hayan terminado: los cierra y recupera los resultados. En este proceso, se utiliza el mutex *isdone* para editar y leer con seguridad las variables *thread_finished*, sin causar *Race Conditions*.

Finalmente, se recorren todas las clases de *thread_data* y se juntan las soluciones devolviendo tres objetos, tal y como se ve en la Fig. 5.5.

5.5. Función RG_GN

En esta función ejecuta el algoritmo RG-GN, expuesto en el apartado 4.5, para cada subregión de la región de interés. Antes de empezar, se extrae la primera correlación calculada para la semilla de dicha subregión, convirtiéndola a matriz (*Ecuación 4.2 y Ecuación 4.3*), y se copian todos los parámetros de la clase *inputs*. Se hace esta copia, para que cada *thread* pueda acceder al valor de estos parámetros sin preocuparse si otro *thread* está accediendo o editando la clase *inputs* en el mismo momento. Esta copia tiene un impacto mínimo en el consumo de memoria total, debido a que todos estos parámetros son relativamente ligeros.

Para ayudar a la comprensión, la Fig. 5.6 ilustra en un diagrama de flujo el proceso que se detalla a continuación.

En primer lugar, se obtiene el *subset* de la iteración guardado en la variable *next_subset* (en el caso de ser el primer *subset*: *next_subset* se obtiene del único elemento en la cola de *subset_queue*). Con este *subset,* y a partir de las matrices de derivadas, se construye la matriz hessiana y se guarda su descomposición de Cholesky en la variable *L;* además de la transpuesta en la variable *L_t*. Se guarda la transpuesta en su propia variable, para evitar hacer este cálculo en cada iteración sobre el mismo *subset*.

Una vez construidas la matriz de deformaciones, la matriz del subset de referencia y la matriz hessiana: se procede al bucle que iterará sobre el *subset* en cuestión hasta encontrar la matriz de deformación óptima.

En cada iteración se accede a 3 funciones que, al final, calculan una matriz de deformaciones W más cercana al óptimo. El bucle termina cuando la norma del vector $P_{tt}^{(n)}$, guardada en la variable NORM, sea más pequeña que el error admisible o cuando se llega al máximo número de iteraciones. Las 3 funciones que se llaman son las siguientes:

1. ***get_warped_sub():*** Esta función deforma el subset de referencia desde la imagen de referencia a la imagen actual, según la matriz de deformaciones $W_{tw}^{(n)}$, guardada en la variable *W*. Una vez deformados, se encuentra sus valores en escala de gris en la imagen actual y se devuelven a una matriz llamada *warped_sub*. Esta sería el equivalente al *subset (b)* de la Fig. 4.2 *en la página 18*.

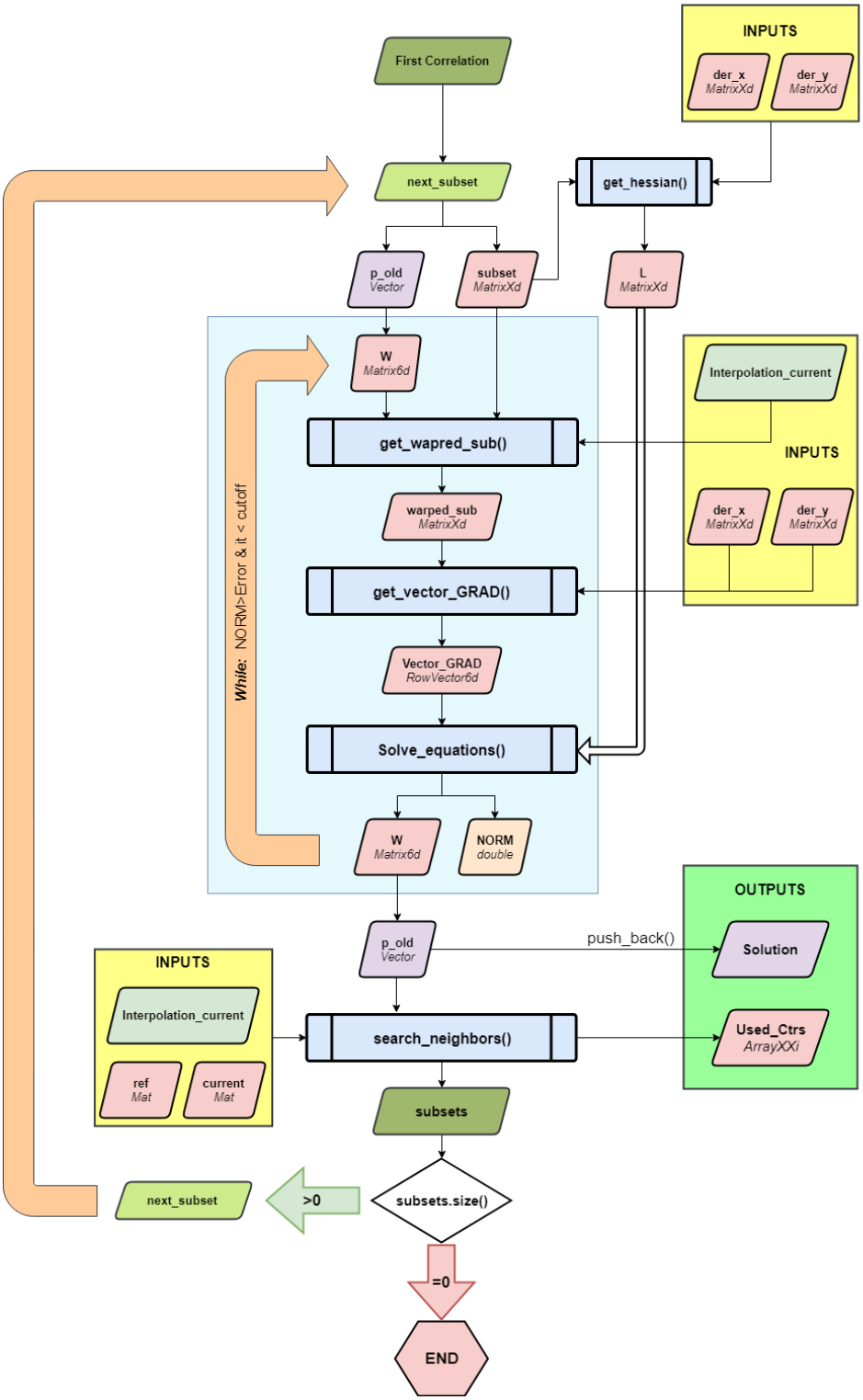

Fig. 5.6 Diagrama de flujo de la función RG_GN_thread(). Las cajas amarillas indican parámetros, de solo lectura, compartidos con otros *threads*; la caja verde claro agrupa las variables de salida de la función.

2. ***get_vector_grad():*** A partir de los dos *subsets* (el de referencia y su deformado según $W_{tw}^{(n)}$, se procede a calcular el gradiente de la serie de Taylor del coeficiente NSSD de la Ecuación 4.13. Este vector se guarda en la variable *Vector_GRAD*, y es un vector de *Eigen3* de dimensiones (1x6).

3. ***solve_equations():*** A partir de la hessiana (y su transpuesta), el gradiente de NSSD y la matriz del subset deformado, se resuelve la Ecuación 4.13 extrayendo el vector $P_{tt}^{(n)}$ y su norma. A continuación, de la Ecuación 4.14 se extrae la matriz de deformación de la siguiente iteración $\left(W_{tw}^{(n+1)}\right)$.

Una vez encontrada la solución optima $P_{tw}{}^{*}$ para el *subset*, siguiendo el algoritmo RG, se llama a la función **search_neighbors()**. En esta función se encuentran los centros vecinos del *subset* en cuestión y se mira si estos cumplen las dos condiciones necesarias para ser añadidos a la cola: si están dentro de la región de interés (*valid_ctr)* y si aún no han sido analizados.

Si los centros vecinos cumplen las dos condiciones, se construye el *subset* para cada uno de ellos y se aplica la deformación óptima encontrada para el *subset* original. Con los dos *subsets* (el de referencia y el deformado) se calcula el coeficiente NCC (Ecuación 4.4). Una vez calculado este coeficiente, se añade el centro vecino a la cola subsets y se marca con un 1 en su posición de la matriz *UsedCtrs*.

La estructura de la cola de siguientes *subsets* (*subset_queue*) se presenta en la *Ecuación 5.3*:

Ecuación 5.3

$$subset\_queue = \{\ (\{\ x_c, y_c\}, p\_old\ , C_{ZNCC}), (\dots), \dots\}$$

Cada elemento de la lista de *subset_queue,* consiste en una tupla de tres elementos: el centro del *subset* en cola; el vector de deformaciones óptimo de su antecesor, p_old , que se usará como primera correlación; y, el valor del coeficiente NCC relacionado con dicho vector de deformaciones.

Finalmente, el algoritmo mira si el vector *subset_queue* está vacío, y si ese no es el caso, se extrae el elemento con mayor coeficiente NCC. Este elemento se guarda en la variable *next_subset* y se borra de *subset_queue*. En el caso que este vacío se termina el algoritmo y se cierra el *thread*.

5.6. Post Proceso

Una vez que la función DIC termine, se procede a los cálculos del post proceso tal y como se introduce en los apartados 4.5 y 4.6. Recordando la Fig. 5.1, la función DIC está escrita en *C++*, pero esta devuelve un vector de soluciones que se utiliza para calcular los campos de desplazamientos y deformaciones. Estos cálculos, que por abreviar se llamaran cálculos del postproceso, se ejecutan desde *Python*. En la siguiente figura, (Fig. 5.7), se expande la caja de la Fig. 5.1 mostrando en detalle el flujo del postproceso y que funciones se llaman desde *Python* y cuales se han delegado a *C++*.

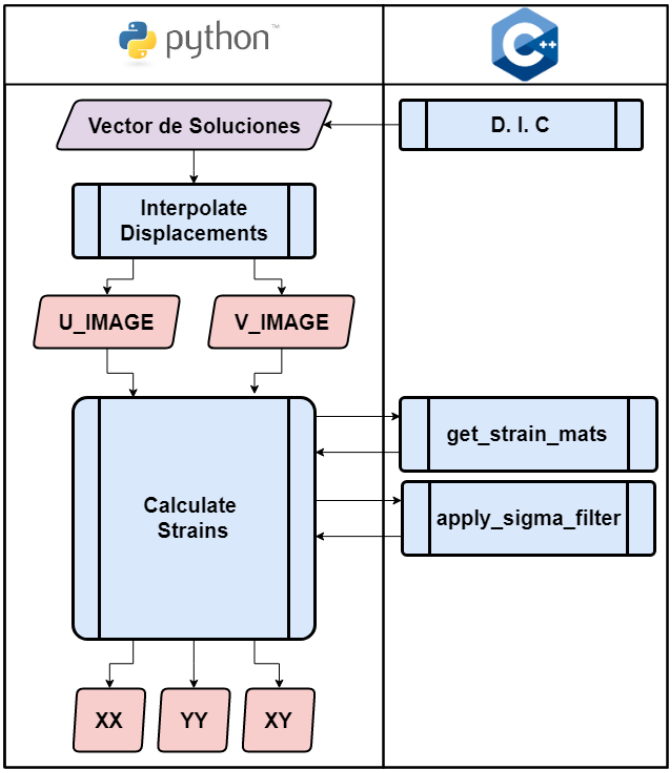

Fig. 5.7 Diagrama de Flujo del postproceso; mostrando la repartición de tareas entre *Python* y *C++*.

La primera función que se llama desde Python es: *Interpolate_Displacementes*. Esta función se encarga de recorrer el vector de soluciones, devuelto por la función DIC, y devolver las matrices del campo de desplazamientos horizontal (*U_Image*) y vertical (*V_Image*). Además, dentro de esta función se utiliza el método *interpolate.griddata* de la librería SciPy **[9]** para interpolar los valores intermedios de los desplazamientos.

Una vez obtenidos el campo completo de desplazamientos se llama a la función *calculate_strains* que devuelve los campos de deformaciones de Green-Lagrange. Dentro de esta función, se llaman a dos métodos escritos en C++: *get_strain_mats* y *apply_sigma_filter*.

Tal y como se ha expuesto en el apartado 4.6, para calcular las deformaciones, se debe iterar sobre cada centro del vector de soluciones, y proceder a hacer unos cálculos matriciales. Como se detallará más adelante, este bucle tarda un tiempo no negligible, y como consecuencia se decide traducir estos cálculos a C++. Estos cálculos se agrupan en la función *get_strain_mats* que devuelve las tres matrices de deformaciones sin interpolar.

Antes de volver a llamar a la función *interpolate.griddata* sobre las matrices de deformaciones, se aplica un filtro para eliminar todos los puntos que estén muy alejados de la media. Esta operación, llamada *apply_sigma_filter*, es opcional, ya que el usuario decide si se hace o no. Finalmente, se llama a la función *griddata* y se interpolan las matrices de las deformaciones.

Por último, se llama a la función *Plot_Graphs* que se encarga de mostrar los gráficos de los campos de desplazamiento, los campos de deformaciones y el recorrido que ha seguido el algoritmo.

6. Proceso de Optimización

En este apartado se expondrá el proceso que se ha seguido para optimizar el rendimiento del CPU y de la RAM del programa. Los resultados numéricos de las siguientes pruebas son susceptibles a la máquina que ejecute el programa; en este proyecto, todas las pruebas se han ejecutado en una Intel Core i7-7500U con 16GB de RAM.

Tabla 6.1: Parámetros del Caso de Estudio de Optimización

lado_subset	31
cnt_space	3
error	10^{-7}
cutoff	50
search_window	300 píxeles
semilla (*x, y)*	(570, 710)
Número de centros por analizar	*49 725*

Como se ha comentado en el apartado anterior, se reserva el uso del *C++* para optimizar los cálculos más pesados del algoritmo. A grandes rasgos, el trabajo ha consistido en optimizar la mayoría de cálculos ya traducidos a *C++* por Antonio Carreras en **[2]**, y solamente pasar a *C++* los cuellos de botella de los apartados que quedaban pendientes de optimizar del trabajo de Marc Guillem Zamora **[1]**. Cabe destacar, que para los primeros 3 apartados (*6.1, 6.2, 6.3*), solamente se evalúa la función DIC del módulo *fastcode*, y no se tiene en cuenta el postproceso.

Antes de empezar, se debe definir un caso de estudio que sirva para comparar los diferentes procesos y para evaluar si los cambios de verdad optimizan el código. Este caso se extrae del *sample13*, de la página web de NCORR en **[3]**. Se extraen las dos primeras imágenes, Fig. 6.1 (a) y (b); se dibuja la región de interés, Fig. 6.1 (c); y, se definen los parámetros del algoritmo en la *Tabla 6.1*.

A continuación, se ejecutan los códigos de los dos proyectos antecesores (la primera versión en *Python* **[1]** y la versión anterior de *C++* **[2]**); además de avanzar los resultados de la última versión optimizada del código de *C++* de este proyecto. En las siguientes figuras, Fig. 6.2, se explora los tiempos de cálculo, por *subset,* y la máxima memoria RAM consumida.

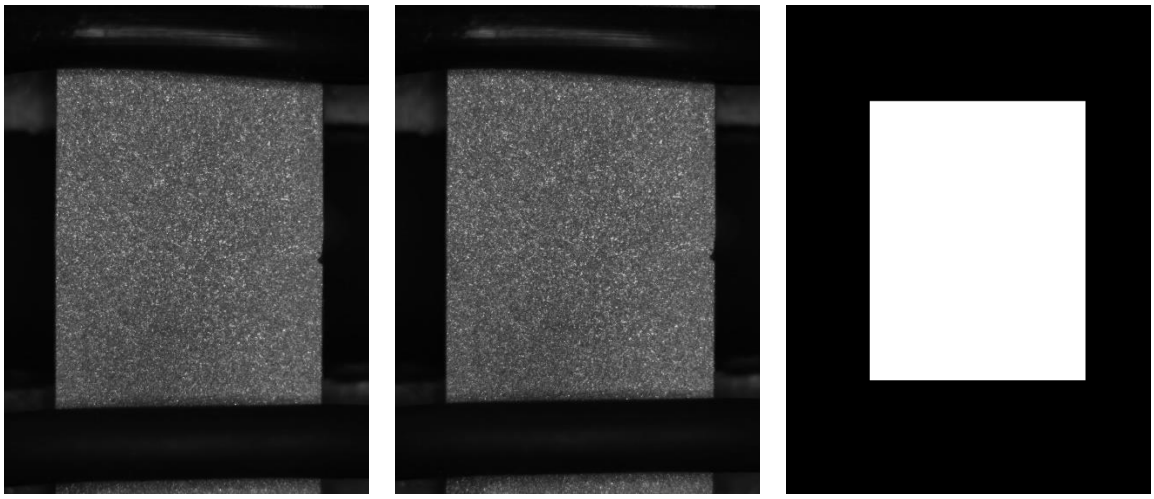

Fig. 6.1 Imágenes del Caso de Estudio para la Optimización. De izquierda a derecha: (a) imagen de Referencia; (b) imagen Actual deformada; (c) Filtro ROI. El rectángulo de la región de interés tiene el vértice superior izquierdo en *(x, y) = (240, 275)* y el vértice inferior derecho en *(x, y) = (850 ,1065)*. Estas coordenadas están en el espacio de coordenadas de la imagen de referencia. Fuente de (a) y (b): NCORR **[3]**

Se reduce el tiempo de cálculo en un 97,3% respecto la versión anterior de *C++* y en un 88,1% respecto la versión original de *Python*. Aun sin ser tan drásticos, los resultados del ahorro de RAM también son mejores en esta última versión que en las dos anteriores.

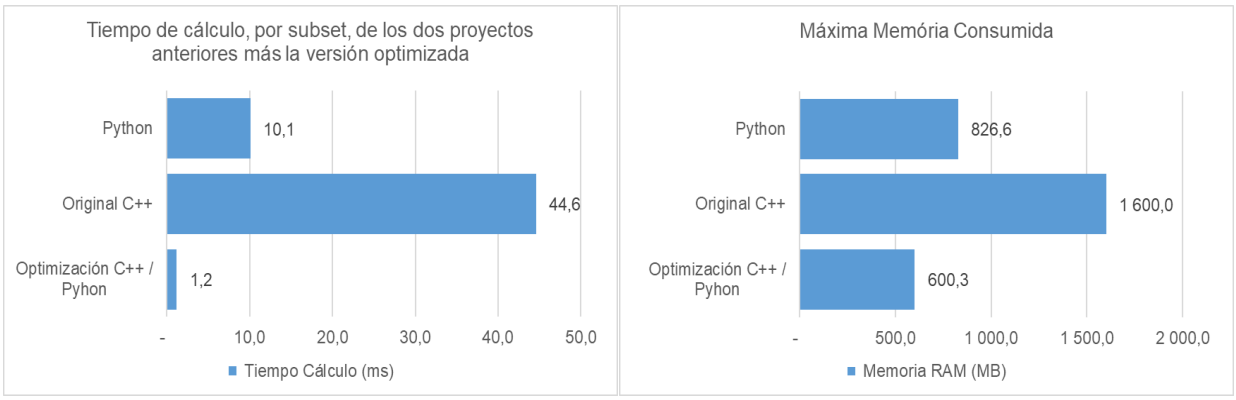

Fig. 6.2 De izquierda a derecha: (a) Comparación del tiempo de cálculo, por *subset*, de los tres proyectos; (b) Comparación de la máxima RAM consumida de los tres proyectos.

El proceso de optimización, de la función DIC, de este proyecto se puede dividir en unas 3 fases:

1. **Limpieza General del Código*:*** En esta primera fase, se recorre el código heredado de Antonio Carreras **[2]**, y se identifican: los cuellos de botella, los cálculos repetidos, las copias de variables, la eliminación de variables de cálculos intermedios y otras buenas prácticas recomendadas para la programación en *C++* **[6]**. Además, se

invierte un tiempo en mejorar la legibilidad del código; restructurar las funciones y los ficheros; y, adaptarlo para que sea más escalable. En este proceso se reduce un 21% el tiempo de cálculo por *subset*.

2. **Cambio de Librería:** Al final de la limpieza general del código, se identifica que los nuevos cuellos de botella están en los cálculos matriciales dentro de los bucles. Se encuentra una nueva librería de *C++* dedicada exclusivamente a los cálculos matriciales: *Eigen3* **[14]**. Se demuestra que esta librería es más rápida que *OpenCv*, y además sus matrices consumen menos memoria. La optimización es incluso más pronunciada cuando se trabaja con matrices pequeñas de tamaño fijo que están completamente optimizadas. Este cambio de librería es el que más impacto tiene sobre la eficiencia del programa, reduciendo en un 90% el tiempo de cálculo respecto la versión del punto 1 de esta lista.

3. **Implementación del *Multithreading*:** como se ha introducido en el apartado 5.4-*Multithreading*, parte de la estrategia de optimización es, dividir el espacio de la región de interés en diferentes partes iguales, y que cada *core* de la máquina itere sobre cada subregión. La reducción del tiempo de cálculo depende enteramente del número de regiones que se quieran paralelizar, y está limitado por el procesador de la máquina que lo ejecute. Cabe advertir, que se puede llegar a congelar el ordenador si se introduce un número demasiado grande de *threads*; y, se recomienda siempre primero mirar las especificaciones del ordenador antes de introducir un número elevado de *threads*.

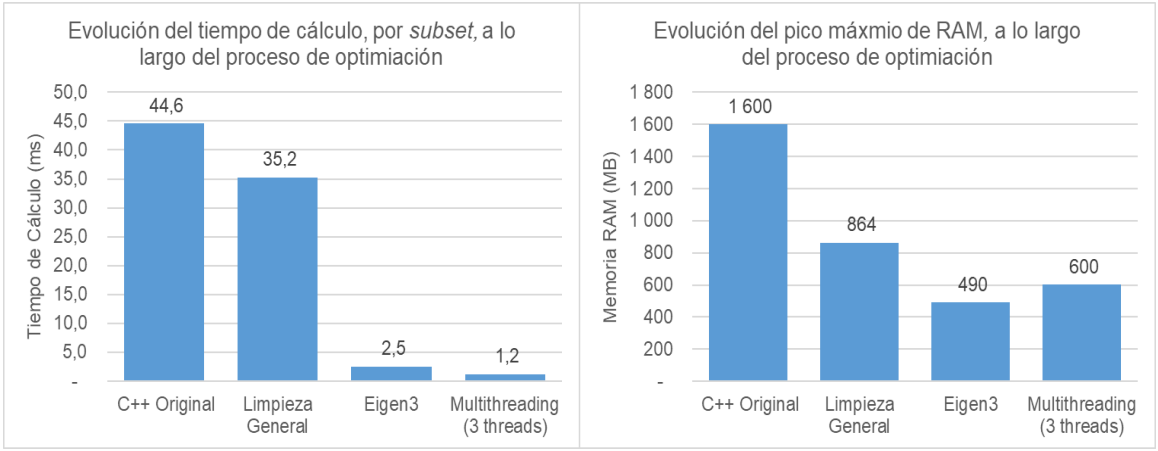

Fig. 6.3: Evolución del tiempo de cálculo (a), y pico máximo de RAM (b), a lo largo del proceso de optimización.

De la figura

Fig. *6.3* (b) cabe destacar que hay un aumento de la memoria RAM consumida al implementar el *Multithreading*. Esto es consecuencia de que cada *thread* necesita una serie de variables para el funcionamiento correcto del algoritmo, y el compilador debe alocar espacio para estas.

6.1. Limpieza General del Código

6.1.1. Vector *P* y matriz *W*

El primer cálculo que se simplifica es la conversión entre la matriz de deformaciones, W_{tw}, y el vector de deformaciones P_{tw}. A modo de recordatorio se vuelven a mostrar las ecuaciones Ecuación 4.2 y Ecuación 4.3, que relacionan las dos variables:

Ecuación 4.2

$$P_{tw} = \left\{ u, v, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial v}{\partial x}, \frac{\partial v}{\partial y} \right\}^T$$

Ecuación 4.3

$$W_{tw} = \begin{bmatrix} 1 + \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & u \\ \frac{\partial v}{\partial x} & 1 + \frac{\partial v}{\partial y} & v \\ 0 & 0 & 1 \end{bmatrix}$$

En el código original de *C++* se efectúa la conversión entre P_{tw} y W_{tw} (o como se escriben en el código*: p_old* y W) más veces de las necesarias, aumentando excesivamente el número de cálculos efectuados. En la Fig. 6.4, se expresa de forma gráfica esta optimización.

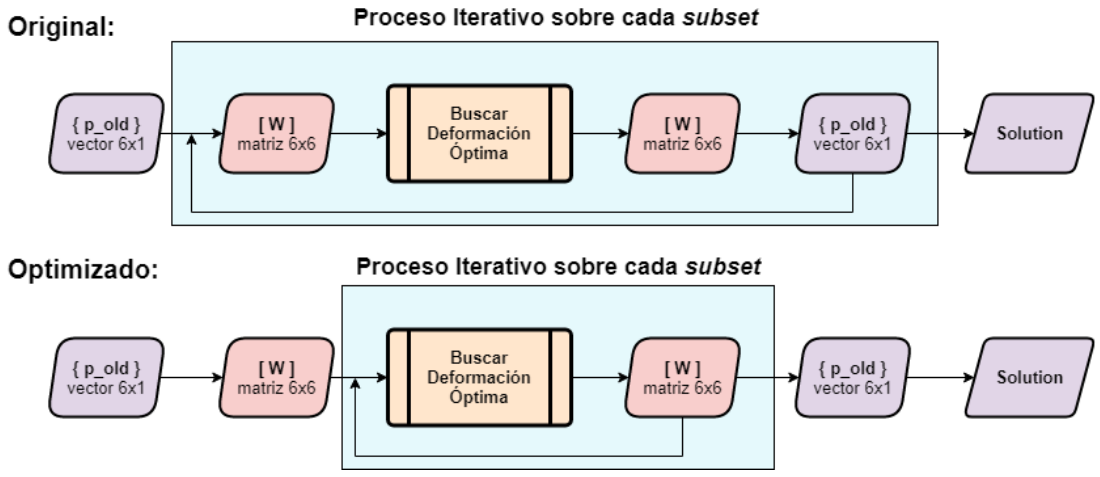

Fig. 6.4 Diagrama de flujo resumiendo la simplificación de la transformación entre el vector *p_old* y la matriz *W*. El recuadro azul representa el proceso iterativo que busca la deformación óptima cada *subset*, y corresponde con el recuadro azul clarito de la *Fig. 5.6*

En la versión original, al principio de cada iteración (n), se transforma el vector encontrado en la iteración anterior, $P_{tw}^{(n-1)}$, al formato matriz, $W_{tw}^{(n-1)}$. A partir de la matriz $W_{tw}^{(n-1)}$, se encuentra la nueva matriz $W_{tw}^{(n)}$, que es más cercana a la deformación óptima, y esta se transforma nuevamente al vector $P_{tw}^{(n)}$. Al comenzar la siguiente iteración $(n+1)$, se vuelve a transformar el vector en matriz y repite el proceso descrito.

En cambio, en la versión optimizada, se hace la conversión del vector $P_{tw}^{(0)}$ a la matriz $W_{tw}^{(0)}$ de antemano, y dentro del bucle sólo se trabaja con la matriz W_{tw}. Una vez se encuentra esta matriz óptima y se sale del bucle y se vuelve a convertir a formato vector P_{tw}, guardándolo en el vector de soluciones.

Con esta simplificación, se eliminan dos cálculos en un bucle que en sí ya está dentro de otro bucle más grande que recorre todos los subsets.

6.1.2. Buscar el máximo NCC

Como se ha explicado en el apartado 5.5, dentro de la función *search_nieghbors*() se encuentran los vecinos del *subset* que se esté analizando, y se añaden a la cola de siguientes *subsets* (*subset_queue*). Al añadir el *subset* vecino a la cola, se pasa: el centro del *subset* vecino; el vector de deformaciones optimo *p_old* encontrado para el *subset* anterior; y el coeficiente de correlación NCC del *subset* vecino deformado según *p_old*. Se presenta a continuación la *Ecuación 5.3* a modo de recordatorio:

Ecuación 5.3

$$subset\_queue = \{ (\{ x_c, y_c \}, p\_old, C_{ZNCC}), (\dots), \dots \}$$

Tal y como se ha explicado en el apartado 4.5, se busca el elemento de la lista *subset_queue* (*Ecuación 5.3)* que presente mayor coeficiente NCC para determinar cuál será el siguiente centro para analizar.

En la versión original del código, para encontrar el siguiente *subset*, se ordena el vector de *subset_queue* según el coeficiente NCC en orden ascendente. Una vez ordenado, se extrae el último elemento (que presenta el máximo coeficiente) y se elimina este de *subset_queue*. En la versión optimizada, en cambio, solamente se busca el *subset* que tenga el mayor coeficiente NCC, sin necesidad de ordenar todo el vector *subset_queue*.

El efecto de esta optimización depende sobre todo del número total de centros a analizar. En

imágenes pequeñas, o con una distancia entre centros muy elevada, la longitud del vector *subsets* se mantiene pequeña y se tarda aproximadamente lo mismo en buscar el máximo como en ordenar el vector entero. En cambio, en imágenes más grandes, o con una distancia entre centros más pequeña, la cola de *subsets* ya es considerable y se encuentra una diferencia de tiempo de cálculo importante.

En el siguiente gráfico, Fig. 6.5, se compara la velocidad de cálculo (en microsegundos) entre ordenar todo el vector o solamente buscar el máximo coeficiente. Se hace una simulación construyendo un vector con la misma estructura de *subset_queue,* pero con valores aleatorios. Además, se va aumentando la longitud de este vector, con el fin de remarcar la diferencia de velocidad de cálculo en vectores largos.

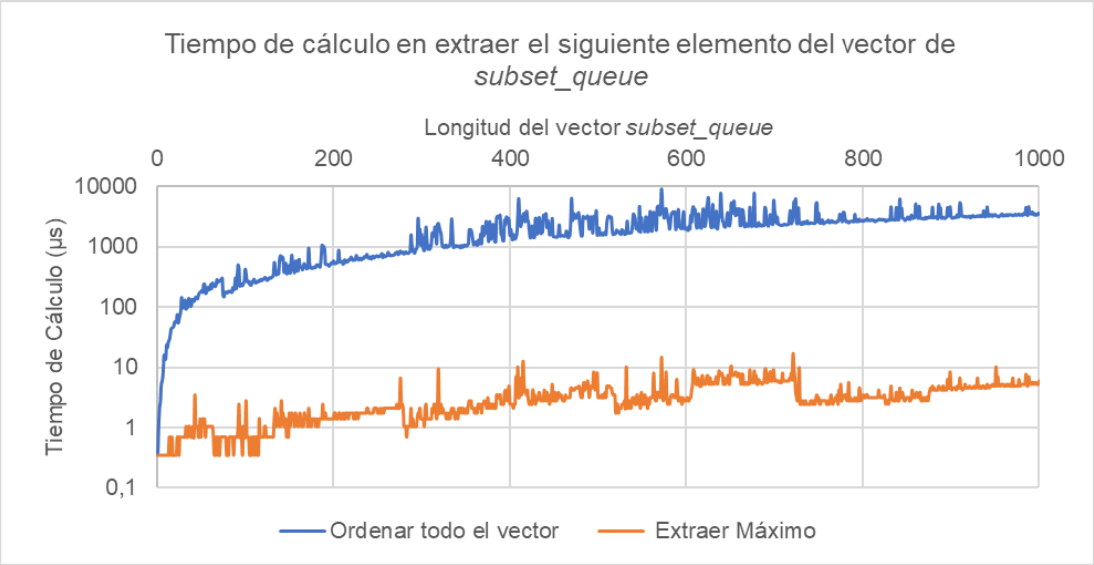

Fig. 6.5 Comparación del tiempo de cálculo entre los dos métodos expuestos para extraer el siguiente *subset*. El eje de ordenadas está en escala logarítmica ayudando a visualizar la diferencia entre los dos métodos.

En efecto, como se puede observar en la Fig. 6.5, enseguida que el vector empieza a ser más largo el tiempo de cálculo en ordenar el vector entero aumenta hasta 3 órdenes de magnitud. En cambio, con el método de extracción del máximo, el tiempo de cálculo se mantiene en el mismo orden de magnitud (entre 1 y 10 microsegundos) incluso con vectores largos.

6.1.3. Borrar Variables

La estrategia seguida para reducir el consumo de RAM es relativamente sencilla: borrar las variables que ya no se utilicen. En *C++* hay diversos métodos para llevar esto a cabo:

- Usando el comando *delete*. Esto solo se puede usar con objetos alocados en memoria dinámica; y para ello, primero deben haber sido inicializados usando el comando *new*. Esta estrategia ensuciaría bastante el código, y se deberían borrar todas las variables una a una.

```
string* x = new string();
//...
delete x;
```

Fig. 6.6 Ejemplo de *new/delete*

- Usando *clear()*. Si se llama al método *clear()* de una variable, se borra todo el contenido dentro de la variable sin borrar la variable en sí. Ej.: se llama *clear()* a un *string*; el contenido del *string* desaparece, pero el *string* en si sigue existiendo, solo que sin contener nada. Esta estrategia es un poco más limpia que el uso de *delete*. En este proyecto se usa en algunos puntos para borrar el contenido de vectores muy largos.

```
string x;
//...
x.clear();
```

Fig. 6.7 Ejemplo de *clear()*

- Limitar el *scope*. Esta es la estrategia que más se usa en este proyecto. Consiste en inicializar las variables en un *scope* inferior, de tal modo que estas se borran cuando el *scope* se acaba.

```
// en main
{
	string x;
	//...
	//utilizar x
} //x se destruye aquí
```

Fig. 6.8 Ejemplo de limitar el *scope*

El ejemplo más claro de limitar el *scope* en el programa, es a la hora de calcular las matrices de derivadas e interpolación. Tal y como se explica en el apartado 4.3, las matrices de interpolación de la imagen de referencia se utilizan solamente para calcular las matrices de derivadas. Por lo tanto, una vez calculadas las matrices de derivadas, se pueden borrar las matrices de interpolación de la imagen de referencia.

En la Fig. 6.9 se presenta este ejemplo del control del *scope*. Como solo se desea guardar los dos objetos de derivadas *(der_x y der_y)* se inicializan estos en el *scope* local del *main*. Entonces, se crea un *scope* inferior donde se inicializa el objeto *interpolation_reference* y se pasa a la función *interp*(). A continuación, se llama a la función *derivative*() que modifica las matrices de derivadas pasadas como parámetros. Una vez *der_x* y *der_y* están calculadas, se cierra el *scope* inferior y se destruyen todas las variables inicializadas dentro (en este caso se borra *interpolation_reference*).

```
MatrixXd der_x(reference.rows, reference.cols);
MatrixXd der_y(reference.rows, reference.cols);
{
	Interpol_Mats interpolation_reference;
	interp(reference, interpolation_reference);

	derivative(
		reference.rows, reference.cols,
		interpolation_reference,
		der_x,
		der_y);
}
Interpol_Mats interpolation_current;
interp(current, interpolation_current);
```

Fig. 6.9 Limitar el *scope* para borrar las matrices de interpolación de la imagen de referencia

La sección de código presentada en la figura Fig. 6.9, es responsable de liberar un 50 % de la memoria RAM respecto el código original de *C++*, en el que no se eliminaba la variable *interpolation_reference*. Esta reducción del consumo RAM, se debe a que las matrices de interpolación son, con diferencia, el objeto más pesado del programa. La evolución de este consumo RAM a lo largo del programa optimizado, se representa gráficamente en la Fig. 6.10.

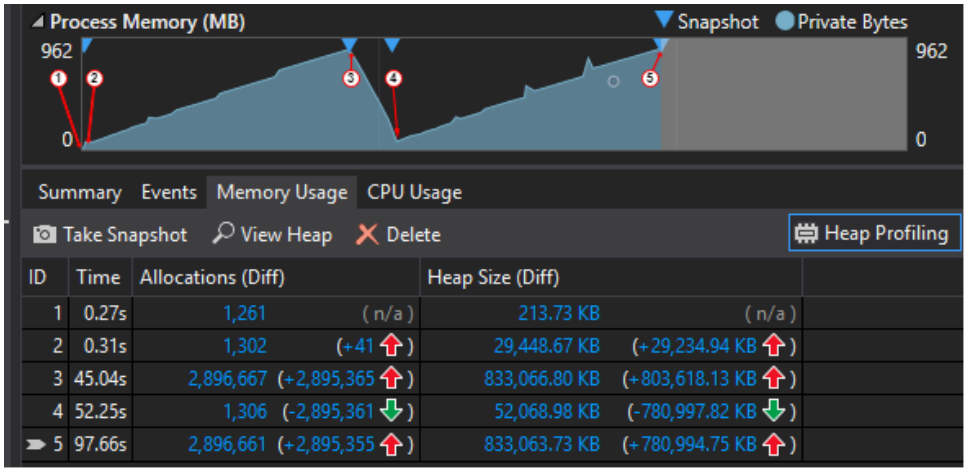

Fig. 6.10 Memoria RAM consumida por el programa en diferentes puntos. Gráfico construido con la herramienta de Diagnósticos de *Visual Studios 2019* **[5]**. Los puntos dentro de los círculos rojos corresponden con los puntos de la columna *ID*.

El punto 3 de la Fig. 6.10 corresponde con el cálculo de las matrices de interpolación y de derivadas; donde el objeto *interpolation_reference* ocupa unos 780 000 KB, que se liberan una vez se obtienen las matrices de derivadas. Si no se borran las matrices de interpolación de la imagen de referencia, el programa llegaría a consumir unos 2GB de memoria RAM.

6.1.4. Reestructuración de ficheros de C++

A diferencia de los tres apartados anteriores, este apartado está enfocado más a incrementar la legibilidad y el orden del código; y no tanto a la optimización de este.

Por el lado de C++, se combinan y renombran ficheros hasta tener los presentados en la siguiente lista. Cada fichero fuente (*.cpp*), con excepción del fichero principal *module.cpp*, tiene asociado su correspondiente fichero de encabezamiento (*.h*) del mismo nombre.

- *module.cpp*: Fichero principal, se definen las funciones que se exponen a *Python*.
- *Binding.h:* Fichero donde se vincula *C++ y Python* mediante la librería pybind11. **[15]**
- *Uitls.cpp/Utils.h:* Se recogen definiciones de objetos (*typedef*) de *C++* de uso general en el programa, y funciones de utilidad general.
- *NCC.cpp/NCC.h*: Define la función que calcula el coeficiente NCC de dos subsets.
- *PreComputations.cpp/PreComputations.h*: Recoge todas las funciones necesarias para los cálculos anteriores al algoritmo DIC.
- *RG_GN.cpp/RG_GN.h*: Recoge todas las funciones para preparar y ejecutar el algoritmo RG_GN.

6.2. Cambio de Librería

Una vez terminada la limpieza del código, se procede a buscar los cuellos de botella del programa. A tal fin, se usa la herramienta de Diagnósticos de *Visual Studio 19* **[5]** que clasifica las diferentes funciones según su consumo de CPU. En las figuras presentadas a continuación, se explora una versión del programa diferente a la presentada en el apartado *5-Estructura del Código*. Esto es debido a que el diagnóstico se ejecuta sobre una versión más antigua que: no tiene la estructura de entorno *multithreading* implementada; donde las matrices de interpolación se guardan directamente en un vector; y, está escrita solamente en C++, sin ninguna función del postproceso. Aún y así, la parte importante que se explorará, sí que conserva la estructura de la Fig. 5.6 de la página 36.

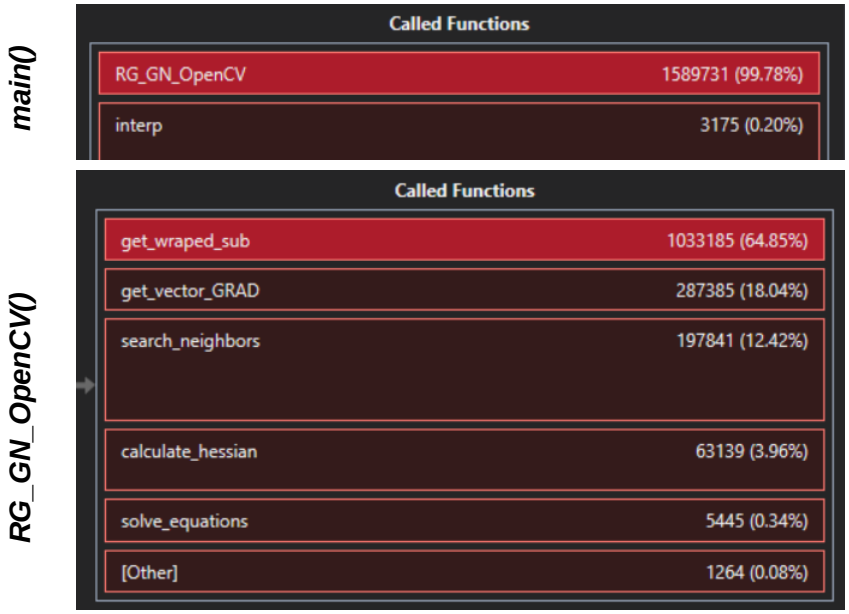

Fig. 6.11 Árbol de Diagnósticos, *Visual Studio 19.* De arriba abajo: (a) Funciones dentro del *main.cpp*; (b) Funciones dentro de *RG_GN_OpenCV(…).* La función *RG_GN_OpenCV* es la misma que RG_GN del apartado *5.5*.

Se observa en la Fig. 6.11, que la función que consume más recursos es, con diferencia, el algoritmo *RG_GN*, que encuentra el vector de deformación óptimo para cada *subset*. Dentro de la función *RG_GN* se obtienen los consumos de CPU de cada subfunción, siguiendo la nomenclatura presentada en la Fig. 5.6 del apartado *5.5-Función RG_GN.*

A continuación, para detallar que partes ralentizan el programa, se entra en las tres funciones que consumen más recursos de la Fig. 6.11(b): *get_warped_sub, get_vector_GRAD* y *search_neighbors*.

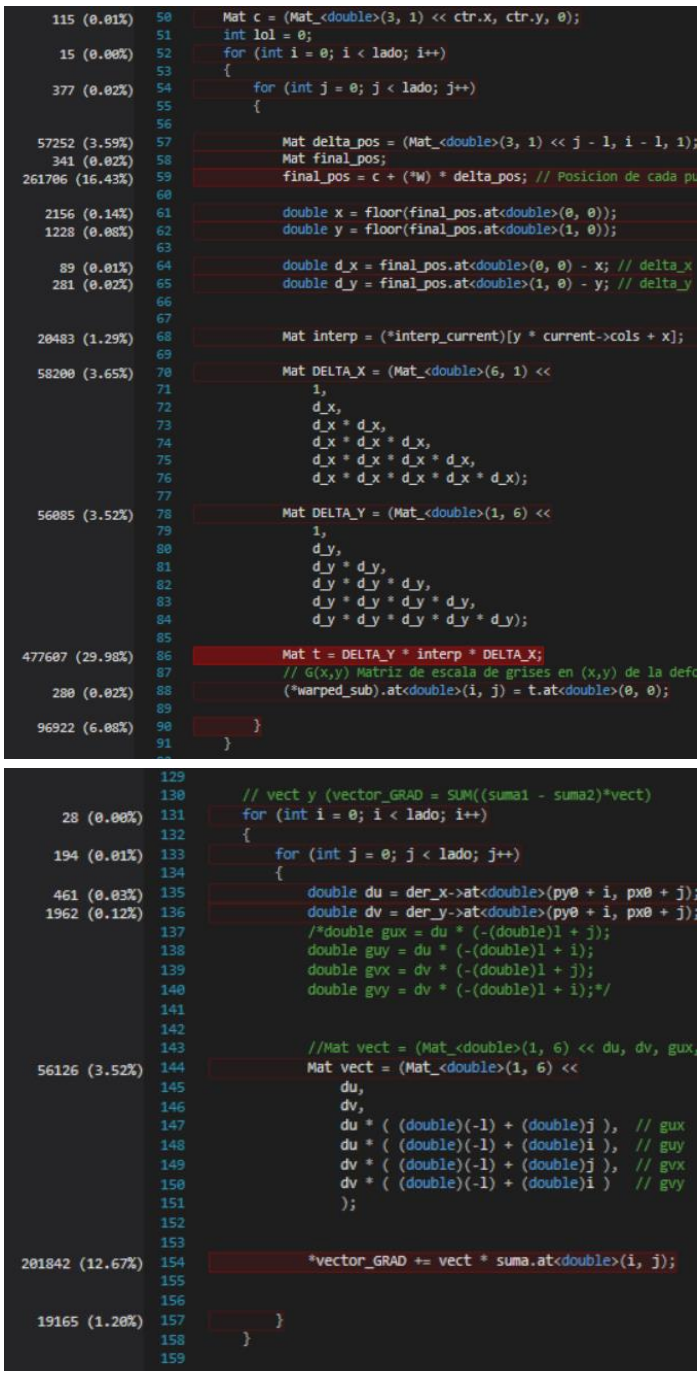

Fig. 6.12 Diagnósticos *Visual Studios* en busca de los cuellos de botella. De arriba a abajo: (a) Diagnostico de *get_warped_sub*; (b) Diagnostico de *get_vector_GRAD*

La parte más costosa de la función *search_neighbors* es cuando se llama a *get_warped_sub* para encontrar el primer coeficiente de correlación; por esta razón, a continuación, solamente se explora las funciones: *get_ warped_sub* y *get_vector_GRAD.*

En la Fig. 6.12, se presentan las líneas del código que consumen más recursos según la herramienta de *Visual Studios*. Se observa que los cálculos más costosos se encuentran en:

- La línea 86 de Fig. 6.12 (a); 29,98% de recursos CPU
- La línea 59 de Fig. 6.12 (a); 16,43% de recursos CPU
- La línea 154 de Fig. 6.12 (b); 12,67% de recursos CPU

Estos tres cálculos consisten en productos, sumas y acceso a matrices. Su optimización tendría un gran impacto, ya que son funciones que se llaman con mucha frecuencia, ya que se encuentran dentro de tres bucles: el que cambia de *subset*, el que itera sobre este *subset* y el que cambia de píxel dentro del *subset*.

Estos tres cuellos de botella son cálculos matriciales con funciones propias de *OpenCv*. Debido a esto, se busca una nueva librería de *C++* con matrices más eficientes.

Una librería ligera y potente dedicada enteramente a cálculos matriciales es la librería *Eigen3*. Además de proporcionar funciones más eficientes, esta librería es compatible con *OpenCv* y *pybind11*. La compatibilidad con *OpenCv* proporciona métodos sencillos y rápidos para convertir de matrices *OpenCv* a matrices *Eigen3*; mientras que la compatibilidad con *pybind11* permite convertir entre las matrices de *Eigen3,* en *C++,* y los objetos de *NumPy,* en *Python*.

Para demostrar la diferencia de velocidad de cálculo entre las dos librerías, se crea una pequeña simulación que imita la estructura de los cálculos de las Fig. 6.12 (a) y (b). En un proceso iterativo, se inicializan unos vectores y matrices de las mismas dimensiones que los presentados en la Fig. 6.12. De estos objetos se hacen dos copias: una guardada en matrices/vectores de *OpenCv* y otra en matrices/vectores de *Eigen3*. A continuación, se hacen los tres cálculos que constituyen el cuello de botella y se mide el tiempo que tardan. Para evitar irregularidades debidas a otros consumos del CPU en el momento y para imitar el orden de magnitud del algoritmo, se repite esta medición unas 1 050 000 veces. Los resultados del tiempo total se presentan en la siguiente tabla.

Tabla 6.2 Comparación entre *OpenCv* y *Eigen3*. Los tiempos de cálculo corresponden a la simulación, no a la ejecución del programa. El porcentaje de mejora se mide respecto el tiempo original de *OpenCv*.

	OpenCv	***Eigen3***	**% Mejora**
Tiempo (ms)	10 799	428	96,04%

Como se observa en la Tabla 6.2, las matrices de *Eigen3* son drásticamente más eficientes. Esta eficiencia se ve amplificada debido a que las matrices, que constituyen los cuellos de botella, son matrices de tamaño fijo. Como se ve en la Tabla 6.3, las dimensiones de estas matrices son siempre iguales, y no dependen de ningún parámetro de entrada. Como se indica en la documentación de *Eigen3* **[14]**, las matrices definidas con tamaño estático están "completamente optimizadas"; y, se recomienda utilizar matrices fijas, en vez de tamaño dinámico, siempre y cuando las dimensiones sean conocidas, constantes e inferiores a 32x32.

Volviendo a referenciar la Tabla 6.3, se observa que ninguna de las matrices que constituyen los cuellos de botella superan este límite de dimensiones; con la excepción al acceso a la variable suma que es una matriz de dimensiones dadas por la longitud del *subset*. Además, el rendimiento del producto no se ve afectado, ya que solamente se multiplica sobre un valor de esta matriz dinámica.

Tabla 6.3 Dimensiones de las matrices/vectores que constituyen el cuello de botella.

Posición en el código y ecuación	**Variables**	**Dimensiones [fila x columna]**
En la línea 86 de Fig. 6.12 (a) $t = DELTA\_Y \times interp \times DELTA\_X$	*DELTA_Y* *interp* *DELTA_X* *t*	1 x 6 6 x 6 6 x 1 1 x 1
En la línea 59 de Fig. 6.12 (a) $final\_pos = c + W \times delta\_pos$	*c* *W* *delta_pos* *final_pos*	3 x 1 3 x 3 3 x 1 3 x 1
En la línea 154 de Fig. 6.12 (b) $vector\_GRAD \mathrel{+}= vect * suma[i,j]$	*vect* *suma* *vector_GRAD*	1 x 6 Dinámica 1 x 6

Concluyendo que las matrices de *Eigen3* son más eficientes, se cambian todas las matrices que contribuyen al cuello de botella al formato *Eigen3*. Para garantizar la compatibilidad de los cálculos, se utilizan dos funciones propias de *OpenCv* que transforman entre *OpenCv* y *Eigen3*: cv2eigen() y eigen2cv(). Estas funciones tienen relativamente poco impacto y se posicionan en puntos estratégicos en el código para reducir, lo máximo posible, el número de conversiones.

Después de hacer un nuevo diagnóstico, resulta que ahora la función *get_hessian*(), que era la única función que se había dejado en la versión original de *OpenCv*, consume un 35% de los recursos cuando antes solo consumía un 3,96%. A razón de esto, se transforma también la función *get_hessian*() a *Eigen3* y se aprovecha para transformar todas las matrices dentro de la función RG_GN a *Eigen3*. Además, se guardan a formato *Eigen3* las matrices que se deben calcular previamente, como las matrices de interpolación y de derivadas. Una vez esta todo transformado a *Eigen3*, se hace un nuevo diagnóstico y el consumo de la función del cálculo de la hessiana vuelve a bajar, esta vez a un 5,8%.

Debido a que los cálculos de las matrices de interpolación, la primera correlación y las matrices de derivadas no son prohibitivamente lentos, la estructura del cálculo no se altera respecto la versión original de *C++*, y se mantiene el uso de *OpenCv*. Lo único que se cambia, es que, al obtener los resultados finales, se convierten a matrices de *Eigen3*. Además de ser más rápidas, las matrices de *Eigen3* son más ligeras. Estas ventajas de las matrices *Eigen3* se pueden ver en los gráficos de la

Fig. 6.3.

6.2.1. Matrices de Interpolación

En el apartado 5.3 se introduce la clase que almacena las matrices de interpolación y que proporciona un método seguro para acceder a este conjunto de matrices; más adelante, en al apartado 6.1.3, se demuestra que este conjunto de matrices son los objetos que más espacio ocupan en la memoria RAM. En este apartado se explorará que objeto contenedor es el más adecuado para almacenar este conjunto de matrices.

Avanzando el resultado, el objeto que mejor almacena las matrices de interpolación es un vector de matrices. Esta estructura es muy similar a la original de *C++,* pero cambiando el vector de matrices *OpenCv* a un vector de matrices *Eigen3* fijas de 6x6. Se explora otros objetos contenedores de *C++*, pero finalmente se acaba revirtiendo al vector. Se conserva en esta memoria el resultado del experimento "fallido" para advertir a quienes continúen este trabajo: que este camino no dio buenos resultados y el objeto que mejor se adapta a las necesidades es, efectivamente, un vector de matrices *Eigen3* (6x6).

Para comprobar cuál es el objeto óptimo para almacenar las matrices de interpolación, se hace una simulación probando de guardar 100 000 matrices (de dimensiones 6x6) de valores aleatorios. Los objetos contenedores primero deben cumplir dos condiciones: que se pueda acceder a cualquier posición dentro del objeto (*Random Access*) y que se pueda insertar las matrices dinámicamente (método *insert* o *push_back*). Gracias a la lista proporcionada por **[6]**, los objetos probados, que cumplen estas dos condiciones, son los que se exponen en la Tabla 6.4. Cabe destacar, que solamente se han probado objetos de la librería estándar de *C++* (*STD*); debido a que: son compatibles con todo el resto librerías de *C++,* están ya optimizados internamente y son seguros, es decir no presentan comportamientos inesperados.

Tabla 6.4 Memoria consumida y velocidad de acceso de diferentes objetos contenedor

Objeto de C++ (STD)	Memoria (KB)	Tiempo de Acceso (nanosegundos)
vector (Eigen3)	3 425	466
vector (OpenCv)	6 529	704
map	3 789	5 059
unordered_map	4 866	3 440
dequeue	3 448	1 614

Todos los objetos de la Tabla 6.4, con la excepción el vector *OpenCv*, almacenan matrices de la librería *Eigen3*. Se presenta el vector de *OpenCv* en la tabla para demostrar el coste adicional en el consumo RAM de las matrices de *OpenCv* respecto *Eigen3*.

Una vez creados los objetos, conteniendo cada uno 100 000 matrices aleatorias de 6x6, se comprueba el tiempo de acceso medio para cada uno. A tal fin, se recorre do todo el objeto en cuestión (todos son iterables), y se mide el tiempo de acceso a cada matriz contenida dentro. Finalmente, se calcula la media de cada tiempo de acceso para cada objeto, obteniendo así los resultados expuestos en la Tabla 6.4.

Al finalizar el experimento, se concluye que el objeto óptimo para almacenar las matrices de interpolación es un vector contenido de matrices *Eigen3*; ya que es tanto el objeto más ligero respecto al consumo de memoria, como también el objeto con acceso más rápido.

6.3. *Multithreading*

La última fase de optimización de la función DIC es la implementación del *Multithreading*. Un entorno *Multithreading* es una serie de procesos, denominados *threads*, que se pueden ejecutar simultáneamente. En el apartado 5.4, se explica la estructura del código que gestiona la implementación del entorno *Multithreading*, en cambio en este apartado se entra en detalle en el proceso seguido para su implementación y optimización.

Como se ha introducido en el apartado 5.4, el mejor método para paralelizar el programa es dividiendo la región de interés en diferentes partes independientes y que se ejecute el algoritmo RG-GN por separado en cada región **[3] [4]**. Antes de ello, pero, se deben cambiar las estructuras de algunas variables, respecto al código de *C++* original, para facilitar y optimizar la partición del ROI y la ejecución del algoritmo RG-GN en cada *thread*. La preparación de estas variables se cubre en el apartado *6.3.1*; mientras que el algoritmo de partición del ROI se cubre en el apartado *6.3.2.*

6.3.1. Adaptación de las Variables *UsedCtrs* y *ValidCtrs*

Para implementar el *Multithreading* eficientemente, primero se debe modificar dos estructuras de datos respecto el código original: el vector de centros usados y el vector de centros válidos.

En la versión original de C++, se marcan los centros analizados o ya añadidos a la cola de *subset_queue*, en un vector de centros tal y como se muestra en la *Ecuación 6.1.* Donde cada elemento del vector en si es un vector de *int* (números enteros); donde el primer valor es la

coordenada *x* del centro del *subset* y el segundo valor es la coordenada *y*.

Ecuación 6.1

$$UsedCtrs_{(versión\ antigua)} = \{\{ x_{c1}, y_{c1} \}, \{ x_{c2}, y_{c2} \}, \dots, \{ x_{cn}, y_{cn} \}\}$$

Por otro lado, la región de centros validos (*ValidCtrs*) originalmente estaba definida como un objeto *Rect* de *OpenCv*. Como bien indica el nombre, este objeto *Rect* representa un rectángulo que venía definido por dos vértices: uno superior-izquierdo y uno inferior-derecho. Cuando se intentaba añadir un nuevo centro a la cola de *subset_queue*, primero se comprobaba que este no estuviese en el vector de *UsedCtrs*, y a la vez que si estuviese contenido en el rectángulo de centros válidos.

En la versión optimizada, estos dos objetos se representan como *Arrays* de *Eigen3*, de unos y ceros. En la Fig. 6.13, se presenta ejemplos de estos *Arrays*.

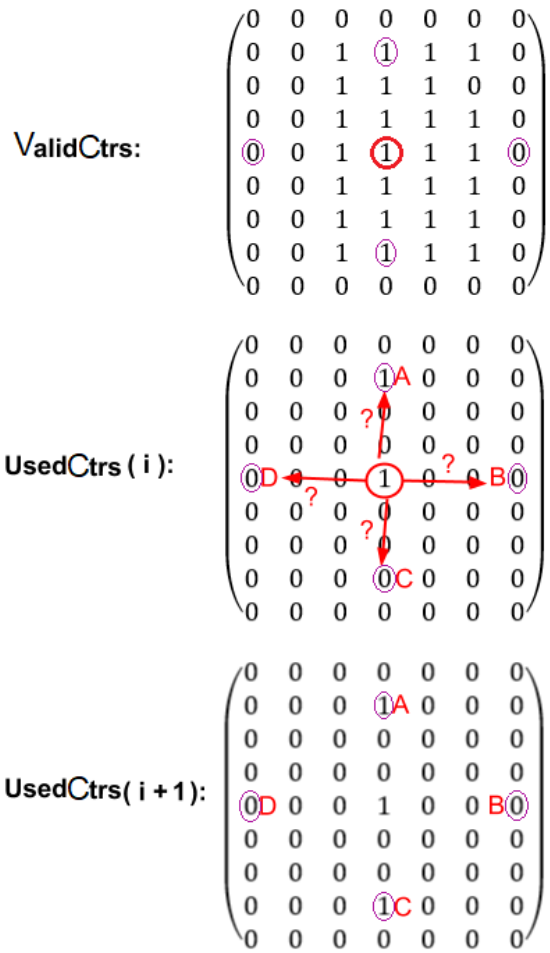

Fig. 6.13 Ejemplo de las matrices *ValidCtrs* y *UsedCtrs*; y de cómo se actualiza *UsedCtrs* entre dos iteraciones.

La matriz de centros válidos, (*ValidCtrs*) marca con "1" la zona donde se puede añadir un centro. Esta matriz se construye a partir del filtro ROI que introduce el usuario. Por otro lado, la matriz de centros usados (*UsedCtrs*) marca con un "1" la posición donde un centro ya ha sido analizado o ya ha sido añadido a la cola de siguientes centros.

El círculo rojo, de la Fig. 6.13, representa el centro del *subset* que se acaba de analizar. A continuación, el algoritmo intentará añadir los centros vecinos a la cola de *subset_queue*. Antes de añadirlos, debe comprobar si los vecinos están dentro de *ValidCtrs* (condición que no cumplen **B** y **D**); y, además debe comprobar que aún no se han analizado estos centros (condición que no cumple **A**).

Al final, solo se añade a la lista de *subset_queue* el centro **C**; y, se marca este en la matriz de *UsedCtrs* con un "1" en su posición.

Representar estas dos variables en formato *Array* tiene dos principales ventajas: el acceso directo a un *Array* de *Eigen3* es más rápido que buscar si este está contenido dentro o no de un vector de vectores y que la representación matricial facilita, a nivel de código, la división de la región de interés.

6.3.2. Partición de la Región de Interés

En esta sección se expone el algoritmo que divide la región de interés, y como se lleva a cabo computacionalmente. El algoritmo de partición de este proyecto está basado en el expuesto

en **[4]**. La división del área de interés consiste en construir un diagrama que indique a qué subregión le corresponde cada píxel. Este diagrama, denominado *Partition Diagram,* se construye como una matriz de enteros, con las mismas dimensiones que la región de interés. Cada valor de esta matriz corresponde con el identificador que distingue cada subregión, que va desde el 0 hasta $N_{threads}-$ 1. Cuando se llame a cada *thread*, este solo recorrerá el área correspondiente a su identificador. Por otro lado, las zonas fuera de la región de interés se marcan con un -1. La Fig. 6.14 muestra un ejemplo de esta matriz al partir un ROI rectangular de 18x18 en tres subregiones (0, 1 y 2).

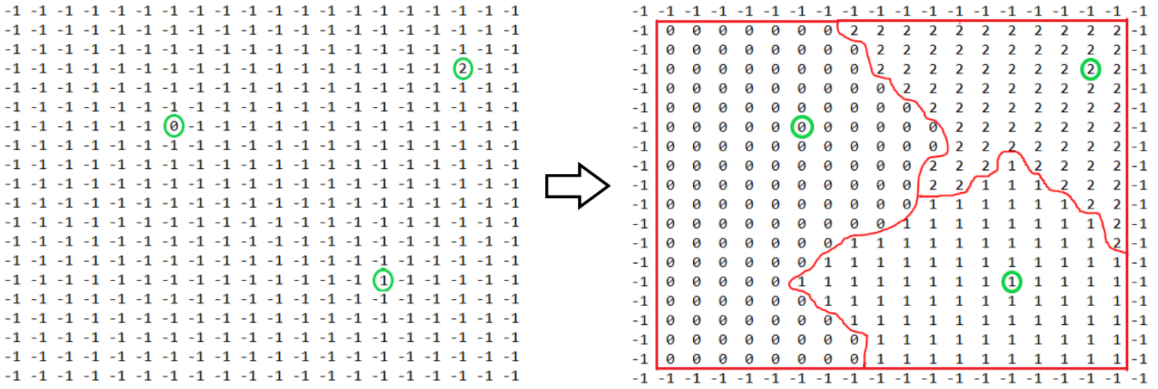

Fig. 6.14 Ejemplo ilustrativo de la forma de un diagrama de partición. De derecha a izquierda: (a) diagrama de partición antes del algoritmo, con todos los valores -1 excepto las semillas; (b) diagrama de partición después del algoritmo, con fronteras rojas pintadas para facilitar la distinción entre las tres subregiones. En los círculos verdes se indica las tres semillas iniciales.

En primer lugar, se inicializa el diagrama de partición con "-1" en todas sus posiciones; excepto en las posiciones de las semillas iniciales, que toman el valor de su identificador. El siguiente paso es aumentar un píxel el área de cada subregión, para que todas las áreas sean aproximadamente iguales. Para decidir a qué píxel se debe propagar la subregión, se utiliza la vecindad de von Neumann de radio 1 **[16]**: que escoge los 4 píxeles adjuntos al centro como candidatos a puntos, tal y como se muestra en la Fig. 6.15. Estos candidatos a punto se añaden a una cola de vecinos candidatos (*neighbor_queue*), junto un identificador adicional que indica la distancia de Manhattan **[17]** (Fig. 6.16) al centro de la semilla original.

El proceso iterativo empieza extrayendo el píxel candidato de la lista *neighbor_queue*, con menor distancia de Manhattan; o, en otras palabras, el píxel más cercano a la semilla original. Si la posición de este píxel es válida, es decir, interior al ROI y aún no ocupada por otra región, se marca el identificador de la subregión en su posición en el diagrama de partición; y, a continuación, se añaden a *neighbor_queue* los vecinos del píxel recién marcado.

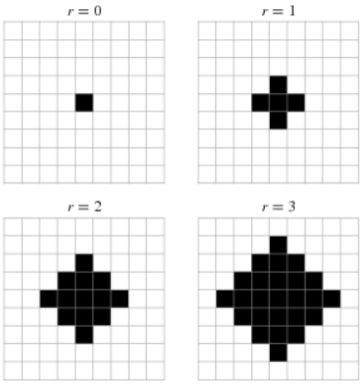

Fig. 6.15 Vecindad de von Neumann para diferentes radios. Fuente: WolframAlpha **[17]**

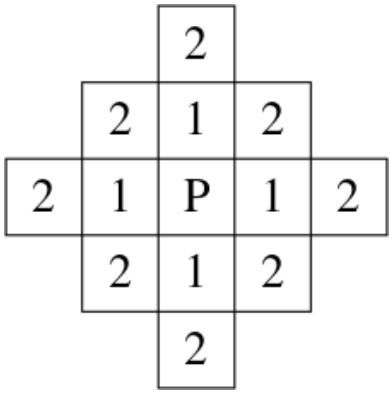

Fig. 6.16 Distancia de Manhattan de los píxeles vecinos respecto la semilla P. Fuente: **[17]**

Antes de añadir un candidato a vecino, se consulta si su posición es posible dentro de la región de interés, y si este valor aún no ha sido adjudicado a ninguna otra subregión. El algoritmo acaba cuando todas las listas *neighbor_queue* de todas las subregiones estén vacías.

Finalmente, se comprueban los resultados usando una región de interés arbitraria dividida en cuatro *threads*, Fig. 6.17 (a). Se dibuja a propósito un agujero central muy irregular para comprobar si esta irregularidad afecta negativamente a la construcción de las subregiones.

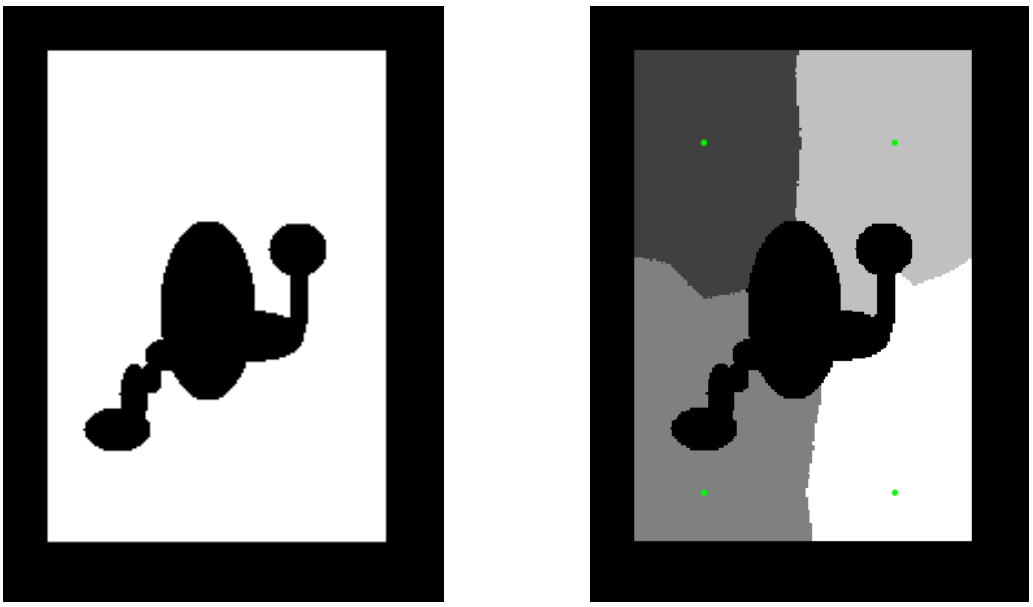

Fig. 6.17 Partición de una región de interés ficticia para comprobar el funcionamiento del algoritmo. De derecha a izquierda: (a) ROI, en blanco indicado la zona de interés; (b) ROI partido, cada color

indica una región y los puntos verdes las posiciones de las semillas iniciales.

Se observa en un análisis visual de la figura Fig. 6.17 (b) que, efectivamente, las regiones de interés creadas son aproximadamente iguales y regulares. Mirando el área que ocupan las regiones, en píxeles, la diferencia entre la región más grande y las más pequeña es de un 7,6%. Se consideran que estos resultados son aceptables para la división de la región de interés.

En **[3]** exponen un método más sofisticado para la división del área de interés. Este método consiste en un proceso recursivo que en cada paso divide las regiones en dos, buscando mediante los momentos de la imagen la mejor dirección para esta partición. En adición, este método coloca automáticamente las semillas en los puntos más internos de las subregiones, buscando los que presenten mejor coeficiente de correlación. Se considera que la complejidad que implica este algoritmo es suficientemente elevada para que no salga a cuenta su implementación; al menos en esta primera iteración del algoritmo de partición. El método presentado en este proyecto es suficientemente bueno para la mayoría de las imágenes y casos con los que se va a trabajar.

6.4. Optimización del Postproceso

La última optimización que se hace tiene lugar en el postproceso. En el proyecto de traducir el algoritmo a *C++* de Antonio Carreras **[2]**, no se llegó a pasar a *C++* la última fase del algoritmo: encontrar el campo de deformaciones a partir del campo de desplazamientos.

Utilizando la estructura híbrida de *Python* y *C++*, se recupera el código de *Python* de Marc Guillem Zamora **[1]** para este último apartado. Como se explica en el capítulo *5-Estructura del Código*, se mantiene la parte que calcula las interpolaciones de los campos de desplazamientos y deformaciones intacta, debido a que estos no sugieren una carga excesiva y además se puede aprovechar gran parte del código ya escrito.

Por otro lado, para calcular las matrices del campo de deformaciones, se debe iterar sobre el vector de soluciones. En el caso que este vector sea largo, el tiempo de cálculo deja de ser negligible, y ya tiene sentido traducir este proceso a *C++* para optimizarlo.

Se efectúan los mismos cálculos que en el código original de *Python*, cambiando algunas partes del código para adaptarlo a las estructuras propias *C++* y *Eigen3*. Una vez se hayan implementado todas las técnicas pertinentes explicadas en los tres apartados anteriores, se evalúa la diferencia del tiempo de cálculo siguiendo el caso de estudio presentado al principio de este capítulo (Tabla 6.1 y Fig. 6.1). En la Tabla 6.5, se presentan los resultados de esta comparación. Cabe destacar que, como el proceso itera sobre el vector de soluciones, se muestra el tiempo de cálculo por elemento de este vector de soluciones, es decir por *subset*.

Tabla 6.5 Resultados de la Optimización del Postproceso

Versión	**Tiempo de Cálculo por *subset* (µs)**
Python	304,5
Python/C++ optimizado	53,1

Esto supone una disminución del tiempo de cálculo de un 82,56%. Esta disminución se debe sobre todo a que recorrer bucles es menos costoso en *C++* y en la eficiencia de la librería de cálculos matriciales *Eigen3*.

7. Interfaz de Usuario y Manual de Uso

En este proyecto se ha desarrollado una simple interfaz de usuario que permite ejecutar el análisis DIC desde diferentes puntos de partida y guardar el progreso en un fichero binario a lo largo de la ejecución. Esta interfaz, se limitará a una ventana de la consola, a la cual el programa imprimirá información y de la cual recibiría las instrucciones del usuario.

Tal y como se explica en el capítulo *5-Estructura del Código*, la interfaz de usuario se construye desde *Python* debido a que tiene herramientas más cómodas para programarla y para guardar y cargar ficheros. A fin de cumplir este objetivo, se usarán dos librerías que vienen incluidas en la descarga estándar de *Python*. Estas se han introducido en el apartado *5.1.1-Librerías Externas*, pero se exponen aquí a modo de recordatorio:

- ***Pickle* [11]**. Esta librería permite guardar todos los tipos de objetos de *Python* a un fichero binario.

- ***Tkinter* [12].** Ofrece unas herramientas para construir una interfaz de usuario completa. En este proyecto, se utilizará únicamente para abrir y seleccionar archivos desde las ventanas de Windows.

A continuación, se seguirá, paso a paso, una ejecución normal del programa, comentando sobre las ventanas e instrucciones que irán apareciendo.

Cuando se inicia el programa desde el ejecutable se abre la consola, abriéndose la siguiente interfaz mostrada en la Fig. 7.1

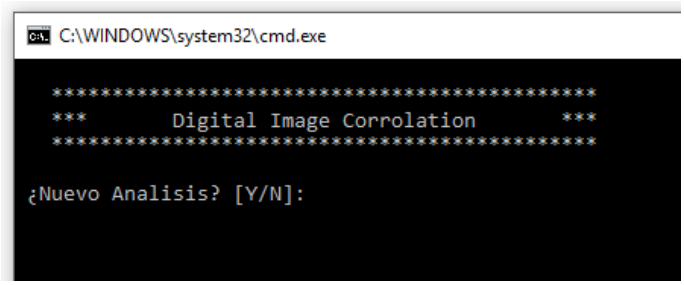

Fig. 7.1 Primera ventana al inicializar el programa.

Lo primero que se pide, es si se está empezando una nueva solución. Si el usuario indica que si: el programa pedirá al usuario que introduzca las imágenes y los datos relevantes para empezar el algoritmo. En cambio, si el usuario indica que no: se abrirá una ventana de Windows para navegar hasta encontrar un fichero binario, de tipo *dicfile* (ej.: *nombre.dicfile*), donde se encuentra la información para retomar el programa desde el punto donde se guardó.

Fig. 7.2 Información General del programa

Para indicar si se quiere empezar un nuevo programa, se puede escribir en la consola (tanto en mayúsculas como minúsculas): **t, true, y, yes, s, si** o **1**. Por otro lado, para indicar falso: **f, false, n, no** o **0**. Si se introduce algún otro carácter, el programa manda un error. Esto es válido para todas las preguntas de si/no a lo largo de la ejecución.

Se continua el tutorial asumiendo que se ha empezado un nuevo análisis, y por lo tanto se muestra en pantalla una serie de explicaciones y recomendaciones sobre los parámetros que se deben introducir. Esta ventana se ve en la Fig. 7.2

Los primeros datos que deben introducir son las imágenes del análisis: imagen de referencia, imagen actual y filtro de la región de interés. El programa abrirá una ventana Windows para que el usuario pueda navegar hasta la carpeta donde se guarden estas imágenes. (Fig. 7.3)

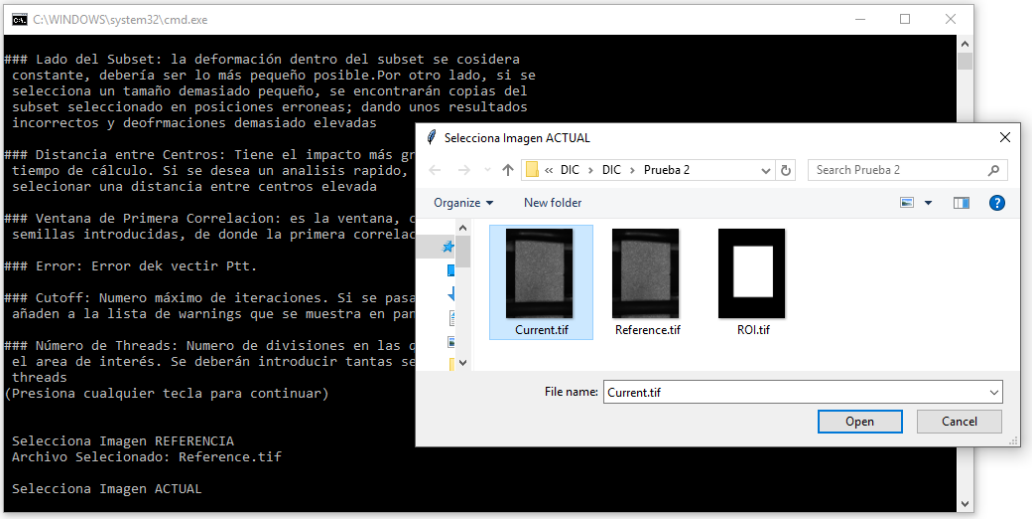

Fig. 7.3 Selección de las imágenes por la ventana de Windows

Una vez seleccionadas las imágenes, se muestran en una ventana para que el usuario pueda consultar si ha seleccionado bien las imágenes que deseaba introducir. (Fig. 7.4).

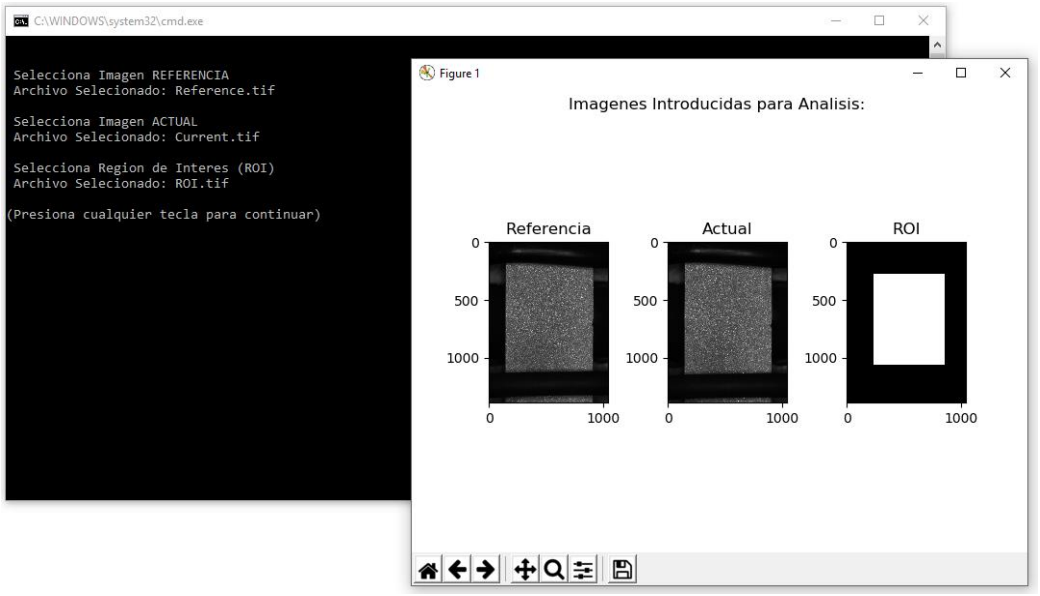

Fig. 7.4 Comprobación de las imágenes seleccionadas.

Seguidamente, se introducen los parámetros del algoritmo y las semillas iniciales. En el caso de indicar que se trabaje con más de una semilla (*n_threads*), el programa devolverá una imagen con la partición del área según las semillas seleccionadas. En la siguiente imagen (Fig. 7.5) se han usado unas semillas al azar, que resulta en una partición muy irregular. Si se desea, se puede indicar que no se está satisfecho con esta partición y se volvería a introducir las semillas.

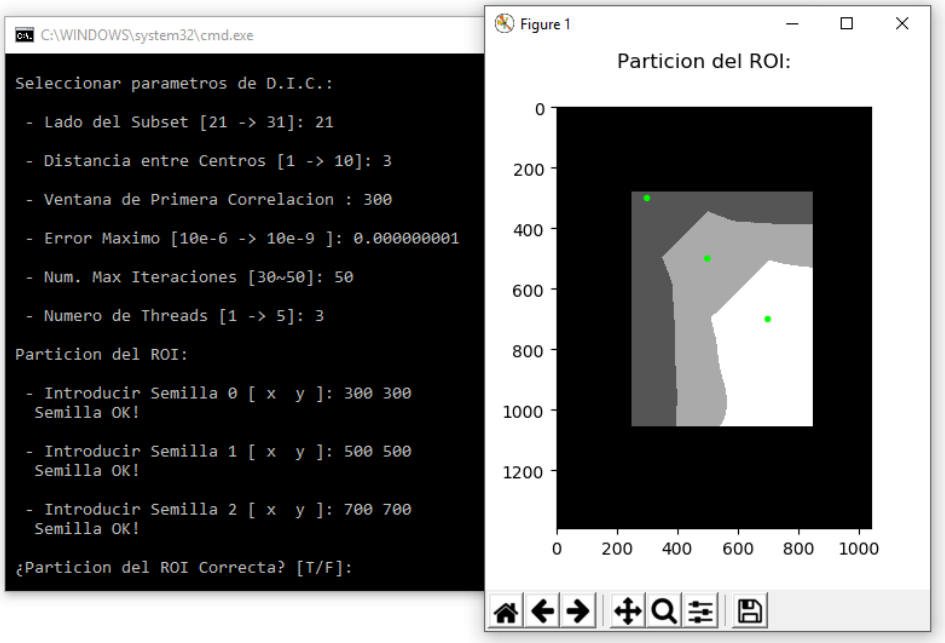

Fig. 7.5 Diagrama de partición según las semillas seleccionadas en la interfaz. Las semillas seleccionadas se muestran con círculos verdes. Los tamaños de estos círculos se pueden cambiar, manualmente en el código, en la función *internal_settings* de la clase inputs. Cada región se dibuja de una sombra distinta para diferenciar las regiones que analizará cada *thread*.

Por último, se seleccionan dos parámetros que solo afectan a la visualización, y se comprueban los datos introducidos. En el caso de indicar que no en la Fig. 7.6, se volvería a empezar esta sección, introduciendo nuevamente todos los datos desde el principio.

```
Seleccionar parametros de interfaz:

 - ¿Mostrar barras de progreso [T/F]: T

 - ¿Mostrar Informacion Adicional en Pantalla (Verbose) [T/F]: T


Informacion Final

 - Lista de Semillas:                        (Se guardan como (y, x) = (i,j) )
   [[300, 300], [500, 500], [700, 700]]

Todo Correcto? [T/F]: T
```

Fig. 7.6 Parámetros de visualización y comprobación de los datos introducidos.

Si se indica que todo es correcto, se procede a ejecutar el algoritmo DIC.

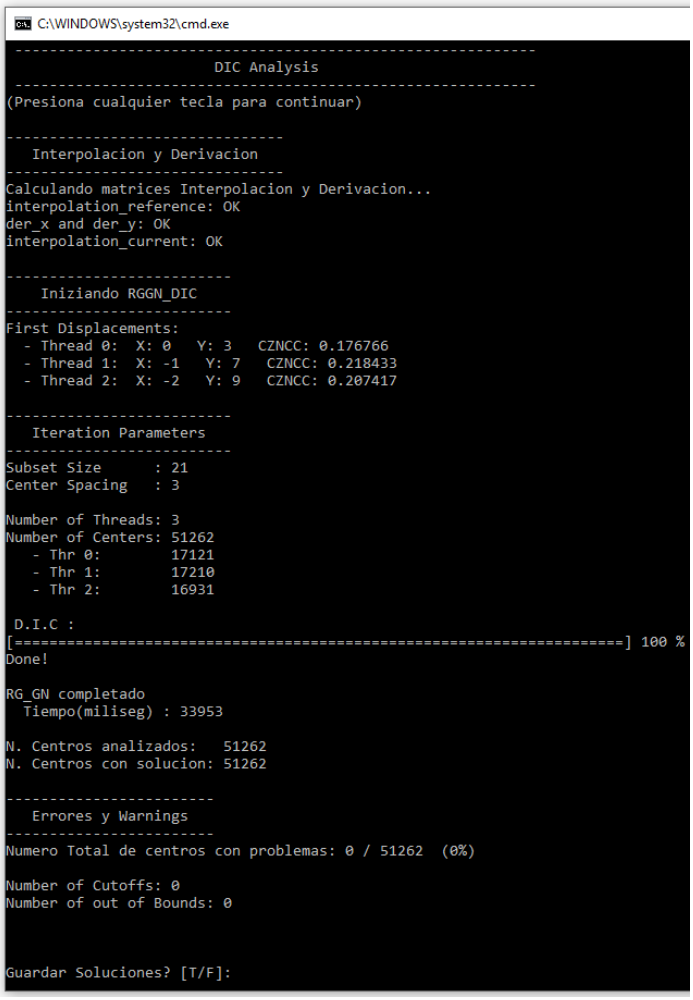

Fig. 7.7 Información del algoritmo DIC que se muestra en la consola a lo largo de su ejecución.

Si se ha indicado que muestre información adicional y las barras de progreso, se vería la siguiente figura (Fig. 7.7).

En la consola se va mostrando el estado actual del programa, y algunos valores de interés como:

- La primera correlación de cada thread, y su coeficiente NCC.
- El número de centros a analizar en todo el análisis, y para cada *thread*.
- Tiempo total de ejecución.
- Número de centros finalmente analizados (pueden ser menos que el máximo, si no se encuentra solución en algunos puntos).
- Número de centros que han encontrado problemas y el tipo de problemas encontrados (*Cutoff* o *Out of Bounds*)

Como al final de cada apartado, se puede guardar los resultados hasta el punto actual en un fichero binario.

En la Fig. 7.8, se muestra un ejemplo de cargar un fichero dicfile. Se guarda el progreso de la Fig. 7.7, y se reinicia el programa. Esta vez se selecciona que no se quiere empezar un nuevo análisis, y se carga el fichero.

Antes de calcular los desplazamientos de Green-Lagrange, se piden dos parámetros adicionales:

- *Strain_window*: La ventana con la que se calculan los desplazamientos.
- Filtro Sigma: Si se desea, se puede aplicar un filtro sigma indicando cuántas desviaciones tipos se quieren mantener.

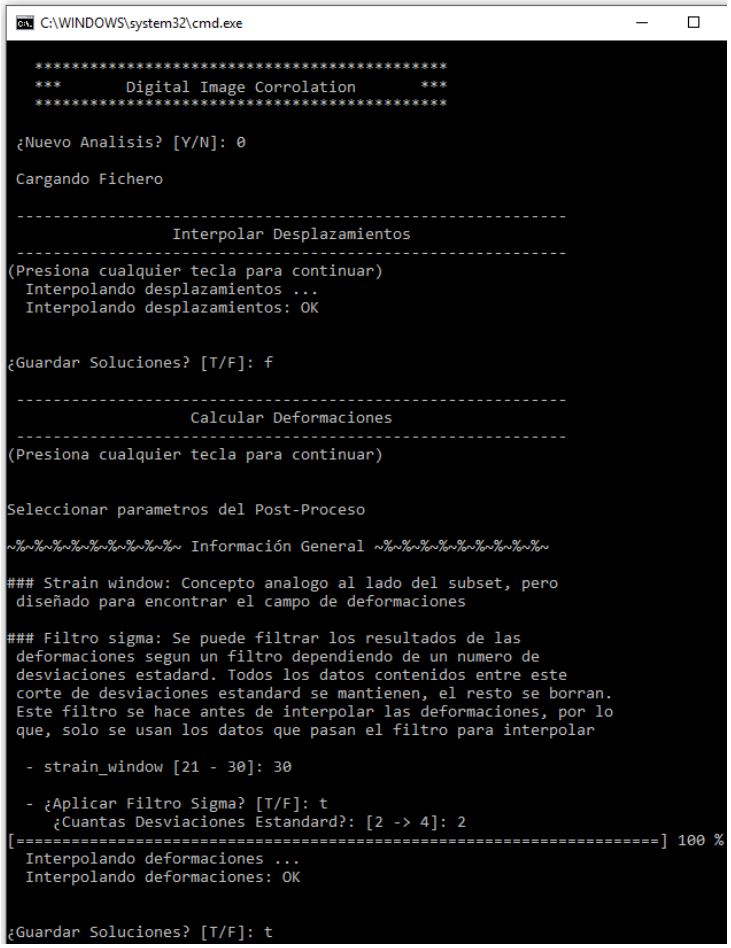

Fig. 7.8 Se carga el fichero donde se había guardado las soluciones del algoritmo DIC, y se ejecuta el post proceso.

Por último, se eligen los gráficos que se quiera dibujar, la conversión de unidades físicas a píxel, y la escala de colores a usar. La interfaz de selección es la Fig. 7.9, mientras que los diagramas que devuelve el algoritmo son las figuras: Fig. 7.10, Fig. 7.11 y Fig. 7.12

Fig. 7.9 Seleccionar los gráficos que se quieran pintar

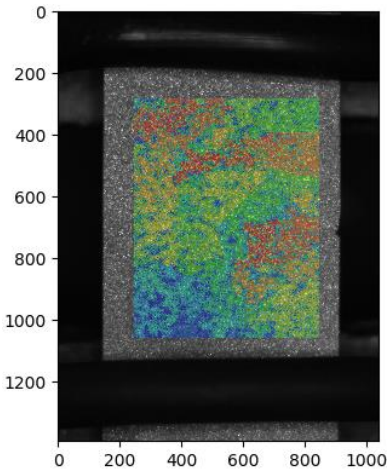

Fig. 7.10 Recorrido del algoritmo RG_GN. Los puntos rojos son los primeros en ser analizados; y los puntos azules los últimos

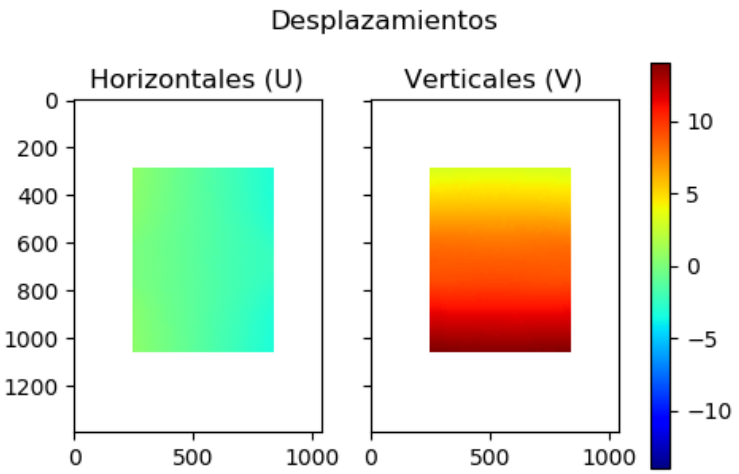

Fig. 7.11 Desplazamientos horizontales y verticales encontrados por el algoritmo DIC.

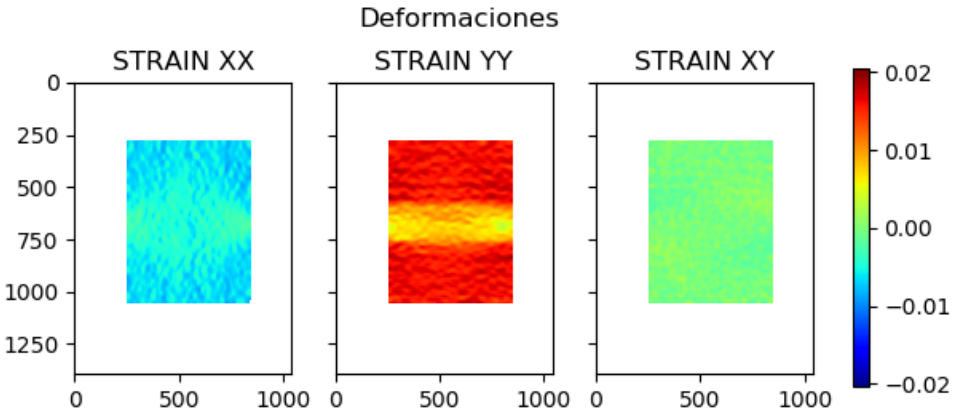

Fig. 7.12 Deformaciones de Green-Lagrange en XX, YY y XY

Tanto en la Fig. 7.11 como en la Fig. 7.12 se centran los resultados alrededor del 0. El máximo siempre se pintará del color rojo oscuro, con el *colormap jet*; el mínimo del color azul oscuro; y el cero de un verde claro.

7.1. Protocolo *dicfile*

A lo largo del programa, se le pregunta al usuario si quiere guardar su progreso. Si este indica que sí, se crea un diccionario de *Python* donde se guardan las clases, las listas y las tuplas que contienen toda información necesaria para retomar el programa más adelante. Una vez construido este diccionario, se convierte, mediante la librería *Pickle* **[11]**, a un fichero binario de tipo *dicfile* (Ej.: *NombreArchivo.dicfile*). A la hora de cargar el fichero, el programa consulta el contenido del diccionario extraído de este, para averiguar qué parte del código debe ejecutar.

La siguiente tabla, Tabla 7.1, muestra los objetos que se guardan en el diccionario a guardar, ordenados según el orden en el que se añaden. Cada objeto del diccionario tiene asignado una llave con la que se puede acceder al contenido de dicha llave, tal y como se muestra en la tabla. En la versión del programa de este trabajo, se guardan entre dos y cuatro objetos, dependiendo del punto en el programa donde se guarde.

Tabla 7.1: Objetos a guardar en el diccionario del fichero binario de tipo *NombreArchivo.dicfile*

Llave del diccionario	Tipo de Objeto	Descripción del contenido de la llave
inputs_class	*<class>*	Clase donde se recogen todos los parámetros, imágenes y opciones introducidos por el usuario al principio del análisis.
dic_solution	*<list>*	Lista que devuelve la función *fastcode.DIC*. Recordando la *Ecuación 5.1*, la lista tiene la siguiente estructura: $\left[\left(\{x_c, y_c\}, \left\{u, v, \frac{du}{dx}, \frac{du}{dy}, \frac{dv}{dx}, \frac{dv}{dy}\right\}, cuenta\right), (\dots), \dots\right]$
displ_tuple	*<tuple>*	Tupla donde se guardan las dos matrices de los campos de desplazamientos horizontales (U) y verticales (V). Estos se guardan ya habiendo interpolado los valores intermedios.
strains_tuple	*<tuple>*	Tupla donde se guardan las tres matrices de deformaciones en el siguiente orden: XX, YY y XY. Igual que los desplazamientos, estas matrices se guardan ya habiendo interpolado los valores intermedios.

A continuación, se muestra un diagrama (Fig. 7.13) que ilustra los puntos del programa donde el usuario puede guardar su progreso, además de mostrar como el programa encuentra al estado que tiene que saltar cuando se carga un fichero *dicfile*.

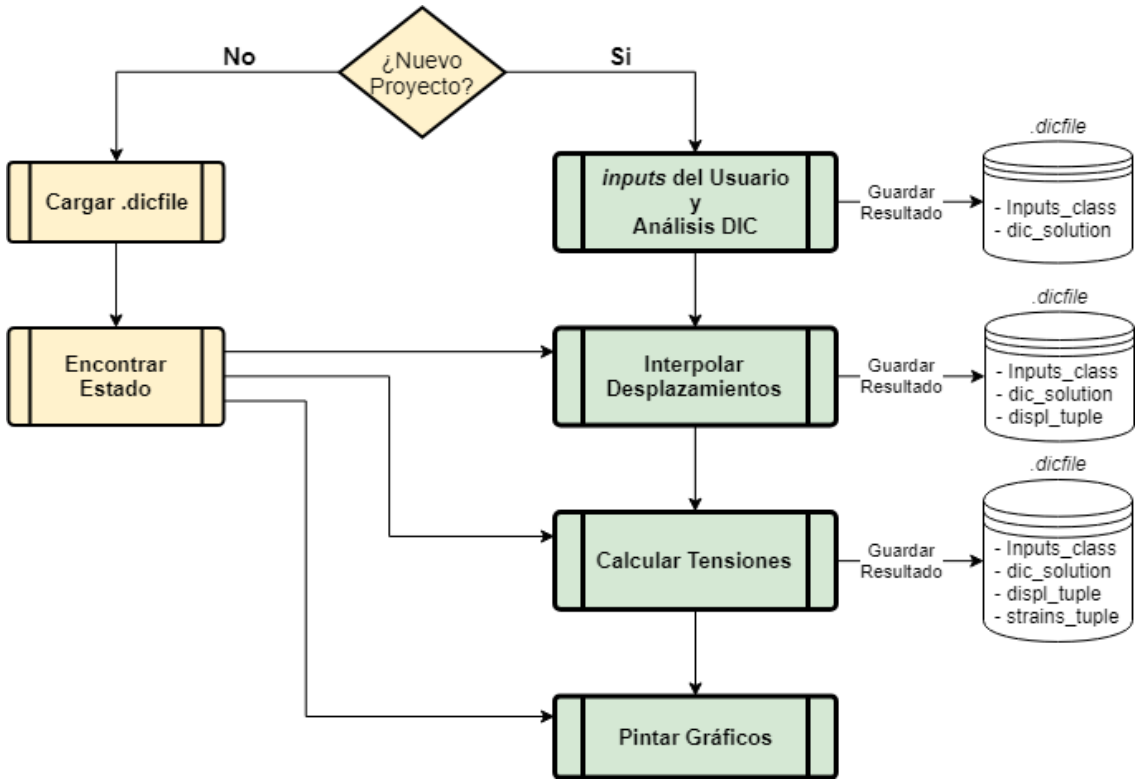

Fig. 7.13 Diagrama de flujo del protocolo *dicfile*. Las cajas verdes representan los diferentes apartados del algoritmo DIC y el postproceso; las cajas amarillas decisiones que toma el programa justo al ser inicializado; y las cajas blancas el contenido del fichero *dicfile* que se guarda en el disco.

Si el usuario selecciona un nuevo proyecto, se ejecuta toda la rama verde de la Fig. 7.13, desde el principio. Al final de cada apartado, si se desea, se puede guardar el progreso en un fichero *dicfile* con el nombre deseado y en la ubicación deseada por el usuario. A medida que avanza el programa va aumentando la cantidad de objetos que se guardan en el fichero binario.

Si el usuario selecciona que no quiere empezar un nuevo análisis, se abre una ventana Windows para que seleccione un fichero *dicfile* de donde cargar la información. Una vez seleccionado dicho fichero, se extrae el diccionario, usando las herramientas de *Pickle*, y consultando las llaves del diccionario se decide a que apartado se debe saltar.

La ventaja de utilizar un diccionario, y consultar sus elementos para saber por dónde retomar el programa es que es escalable. En el casual que en un futuro se añadan nuevos pasos al análisis, y como consecuencia nuevos objetos a guardar, se pueden adaptar bien a la estructura actual, teniendo que cambiar relativamente poco código

8. Comprobación de Resultados

Siguiendo el procedimiento que se encuentra tanto en **[1]** como en **[3]**, se sigue una serie de pasos para comprobar si los resultados obtenidos por el programa son correctos o no. A tal fin, se estudiarán dos casos "artificiales" que ponen a prueba tanto el cálculo de los desplazamientos como el de las deformaciones. Estos casos son:

1. Translación de solido rígido. Se moverá artificialmente (editando la imagen) la imagen de referencia unos ciertos píxeles conocidos. Para un correcto funcionamiento, se espera que los desplazamientos indiquen el número de píxeles desplazados; mientras que las deformaciones deberían ser lo más cercanas a cero posible.

2. Rotación pura. También se mueve la imagen artificialmente unos grados conocidos. En este caso, se espera que las deformaciones también se mantengan próximas a cero.

Los ficheros que se usarán en los dos siguientes apartados se pueden encontrar en **[3]**, entrando en la pestaña "*Aplications" > "Rigid Body Translation & Rotation" > "verification.zip*". En ellos se encuentra unos ficheros de Matlab que se pueden convertir a imágenes siguiendo las instrucciones. En este proyecto se ha utilizado la versión *open source* Octave **[7],** que para este caso funciona igual.

8.1. Translación

La primera comprobación consiste en desplazar la imagen de referencia -4.25 píxeles en la dirección *x*, y -2.75 píxeles en la dirección *y*. Los parámetros de la ejecución se presentan en la siguiente tabla:

Tabla 8.1 Parámetros de translación pura

Lado subset	21		Número de Threads	1
Distancia entre centros	1		Semilla *(x, y)*	(200, 200)
Ventana primera correlación	200		*Strain_window*	30

Error máximo	10^{-9}	Filtro Sigma Número de desviaciones tipo	si 2

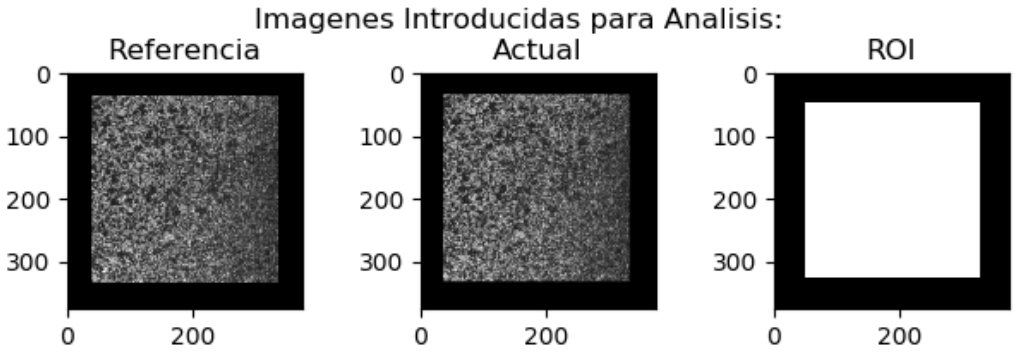

Fig. 8.1 Imágenes del estudio de la translación pura

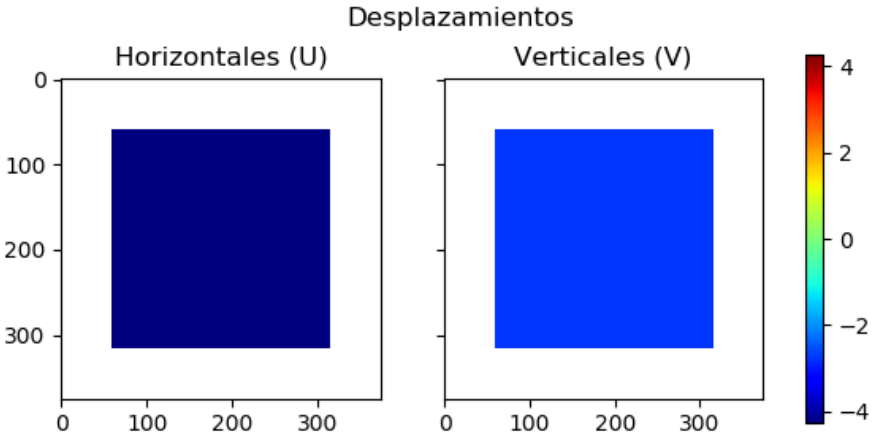

Fig. 8.2 Desplazamientos horizontales y verticales. Los valores que se presentan los desplazamientos son -4,25 píxeles en horizontal y -2,75 píxeles en vertical

En la Fig. 8.2, se ven que los campos de desplazamientos, tanto verticales como horizontales, son uniformes y con los resultados esperados.

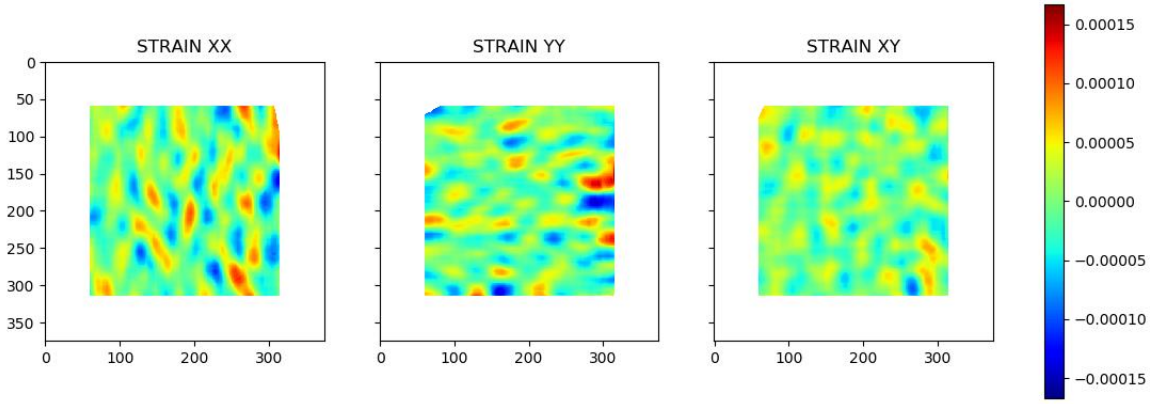

Fig. 8.3 Deformaciones de Green-Lagrange de la translación pura.

Los resultados de las deformaciones, presentados en la Fig. 8.3, muestran resultados cercanos a cero, pero con un error mucho mayor. Mientras que la precisión del campo de desplazamientos es de 10^{-9}, el campo de deformaciones presenta un error de 10^{-4}. Este problema es el mismo que expone Marc Guillem Zamora en su proyecto **[1]** al ejecutar un experimento parecido. Se ha intentado resolver parte de este error añadiendo un filtro sigma, pero lo único que cambia es que los campos de deformaciones son algo más continuos y que se eliminan los valores extremos.

8.2. Rotación

La segunda prueba consiste en rotar la imagen de referencia unos 5 grados en sentido horario. Los datos se extraen de la misma fuente que la prueba anterior y se usan los mismos parámetros presentados en la Tabla 8.1.

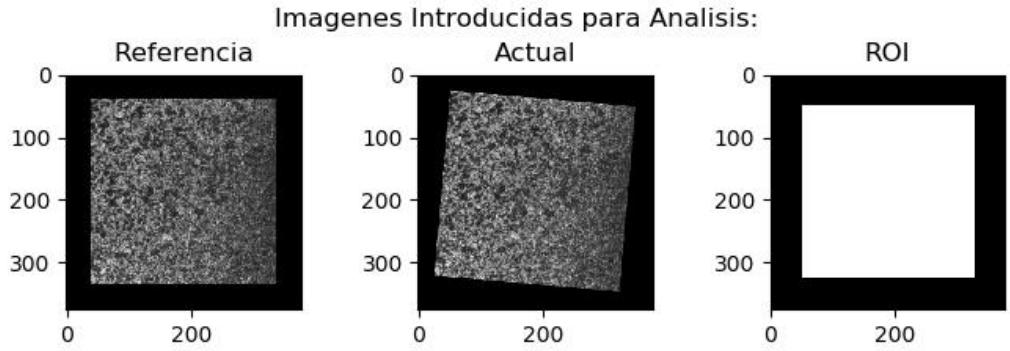

Fig. 8.4 Imágenes del estudio de la rotación pura.

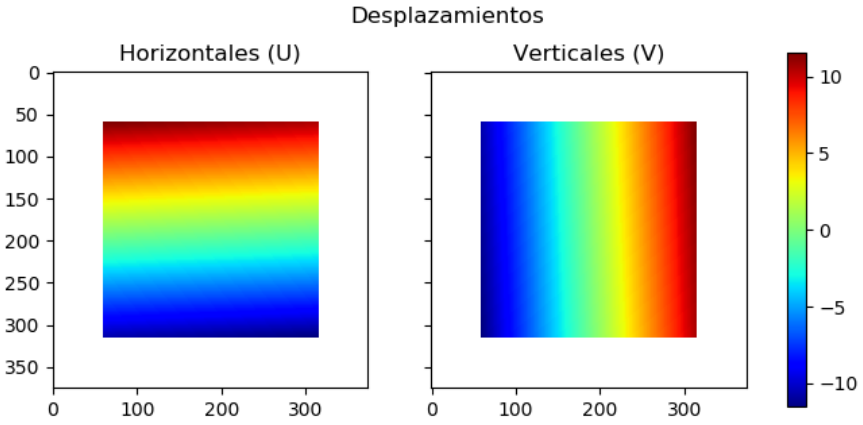

Fig. 8.5 Desplazamientos horizontales y verticales de la rotación pura

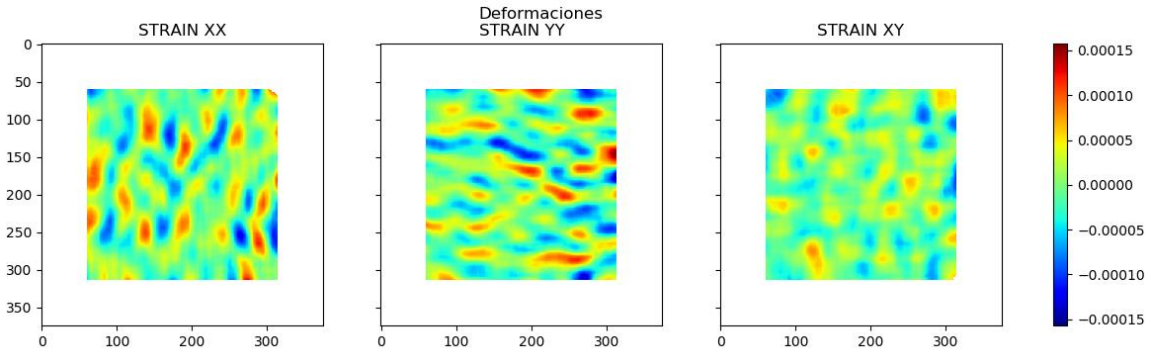

Fig. 8.6 Deformaciones de Green-Lagrange de la rotación pura.

En el estudio de la rotación pura, se tiene el mismo problema sobre el orden de magnitud de las deformaciones. En el caso del trabajo de Marc Guillem Zamora, había un error del orden de 10^{-5}, mientras que los resultados que se ven en la Fig. 8.6 son de un error del 10^{-4}. Viendo estos resultados se concluye que es necesario más trabajo sobre el cálculo de las deformaciones. Se debería comprobar primero si al trasladar las matrices de *Python* a C++ se pierde información por el camino debido a diferencias en la manera de almacenar los datos. Otro posible error es que el cálculo de los mínimos cuadrados de la librería *Eigen3* de resultados diferentes que el mismo cálculo de la librería *NumPy*.

Conclusiones

En este trabajo se han cumplido eficazmente los dos objetivos principales presentados: el aumento del rendimiento del programa respecto a los dos trabajos antecesores e implementar el postproceso en la versión optimizada del programa. Estos objetivos se han cumplido debido a una correcta combinación entre *Python* y *C++* que maximiza las ventajas de cada uno de los lenguajes de programación: mantener el código flexible para la continuación de este proyecto a la vez de una importante optimización en un lenguaje más rígido.

La velocidad de cálculo siempre se ha comparado en solo un *thread* con las otras dos versiones, de esa manera optimizando primero el algoritmo en sí, antes de introducir el entorno *multithreading*, que paraleliza el programa en funciones independientes. Además de una reducción importante en el tiempo de procesamiento, se ha reducido el consumo de memoria total y también se han reducido los picos de memoria máximos que ocurrían al principio del programa.

También se ha demostrado que se necesita más trabajo en el cálculo de las deformaciones, ya que este ha resultado en datos con más error que la versión original de *Python*. En la siguiente lista resume diferentes puntos por donde se podría continuar este trabajo para cumplir los objetivos globales de este proyecto conjunto:

- Comprobar resultados: Se necesita comprobar los diferentes puntos del postproceso, ya sea las funciones de interpolación de los campos de desplazamientos y deformaciones o la función de mínimos cuadrados de la librería *Eigen3* que estén causando el aumento en el error que se observa.

- Mejorar el algoritmo de partición. Como se presenta en **[3]**, existe un algoritmo de partición de la región de interés más sofisticado que el que se usa en este proyecto. Este algoritmo divide automáticamente la región de interés sin necesidad de introducir semillas iniciales. A partir de un algoritmo recursivo, se divide la ROI según los momentos de la imagen; y se encuentran una serie de puntos, estratégicamente escogidos, seleccionando los que presenten mejor correlación inicial.

- Interfaz de Usuario: la implementación de una interfaz de usuario potente y completa facilitaría el uso del programa, ya que este se ejecuta actualmente desde la línea de comandas.

- Multi-Step: En la versión actual, el algoritmo encuentra la correlación solamente entre

dos imágenes. Una mejora sería implementar la posibilidad de introducir una serie de imágenes consecutivas, y ejecutar el algoritmo DIC sobre cada pareja, agrupando finalmente los resultados. Esto permitiría analizar deformaciones más grandes, ya que entre foto y foto consecutiva se cumplirían las condiciones de pequeñas deformaciones.

Agradecimientos

Primero de todo me gustaría agradecer a Marc Guillem Zamora Agustí por empezar el proyecto conjunto del que ha surgido este trabajo, y por su ayuda para entender el algoritmo y el código, además de las sugerencias e ideas que ha propuesto para su realización. Me gustaría agradecer en especial su atención, sobre todo en los meses de verano.

En segundo lugar, me gustaría agradecer a Miquel Ferrer Ballester por la flexibilidad que se me ha dado para trabajar en este proyecto.

Bibliografía

Referencias bibliográficas

[1] ZAMORA AGUSTÍ, MARC GUILLEM*: Development and analysis of a NumPy-based DIC application for strain calculation*, Barcelona, 2018.

[2] CARRERAS LUQUE, ANTONIO: *Contribució al desenvolupament de programari de correlació digital d'imatges (DIC)*, Barcelona, 2019.

[3] J BLABER, B ADAIR, AND A ANTONIOU, *Ncorr: Open-Source 2D Digital Image Correlation Matlab Software.* Experimental Mechanics (2015). Disponible en: <http://www.ncorr.com/index.php>

[4] BLABER, J. *Ncorr* [online]. Atlanta: Georgia Institute of Technology, 2018 [Date of reference: 29 December 2017]. DOI: 10.1007/s11340-015-0009-1.

[5] MICROSOFT, *Visual Studio Community 2019 IDE* (Versión 16.2.3) [Software; Fecha de Referencia: 23 de agosto, 2019]. Disponible en: <https://visualstudio.microsoft.com/vs/>

[6] THE C++ RESOURCES NETWORK, *Standard C++ Library Reference* [online, Fecha de referencia: 18 Agosto 2019], Cplusplus.com. Disponible en: http://www.cplusplus.com/reference/.

[7] JOHN W. EATON, DAVID BATEMAN, SØREN HAUBERG, RIK WEHBRING (2019). *GNU Octave version 5.1.0 manual: a high-level interactive language for numerical computations.* Disponible en : <https://www.gnu.org/software/octave/doc/v5.1.0/>

[8] NUMPY COMMUNITY, *Numpy Reference: Release 1.16.0* [online]. Python Community, 31 enero 2019. Disponible en: <https://docs.scipy.org/doc/numpy-1.16.0/reference/>

[9] HUNTER, J. D., *Matplotlib: A 2D graphics environment.* Computing in Science & Engineering, 2007. (Versión 3.1.1) [Fecha de Referencia: 23 Agosto 2019] Disponible en: < https://matplotlib.org/ >

[10] SCIPY COMMUNITY, *SciPy Reference Guide: Release 1.3.0 [online].* Python Community, 17 mayo 2019. [Fecha de Referencia: 23 Agosto 2019]. Disponible en: <https://docs.scipy.org/doc/scipy-1.3.0/reference/>

[11] PYTHON SOFTWARE FOUNDATION, *Pickle Documentation for Python 3.7* [online]. [Fecha de referencia 28 Agosto 2019]. Disponible en: <https://docs.python.org/3/library/pickle.html>

[12] PYTHON SOFTWARE FOUNDATION, *Graphical User Interfaces with Tk* [online]. [Fecha de referencia 28 Agosto 2019]. Disponible en: <https://docs.python.org/3/library/tk.html>

[13] OPENCV, *Open Source Computer Vision Library,* [Fecha de Referencia: 1 agosto 2019, Versión: 4.1.0]. Disponible en: <https://opencv.org/>

[14] GAËL GUENNEBAUD, BENOÎT JACOB, *Eigen v3* [Fecha de Referencia 23 Agosto 2019]. Disponible en <http://eigen.tuxfamily.org/ >

[15] WENZEL JAKOB, *pybind11 9a19306f documentation*, 2019. [Fecha de referencia 1 Julio 2019]. Disponible en <https://pybind11.readthedocs.io/en/stable/>

[16] WEISSTEIN, ERIC W. *von Neumann Neighborhood.* From MathWorld--A Wolfram Web Resource. <http://mathworld.wolfram.com/vonNeumannNeighborhood.html>

[17] KRISHNAVEDALA, *Manhattan Distance r = 2*, 13 enero 2016. Disponible en: <https://en.wikipedia.org/wiki/Von_Neumann_neighborhood#/media/File:Manhattan_distance_r=2.svg>

[18] PARTLOW, J. *Write C++ Extensions for Python - Visual Studio*. - Visual Studio | Microsoft Docs, 19 noviembre 2018. [online]. Disponible en : <https://docs.microsoft.com/en-us/visualstudio/python/working-with-c-cpp-python-in-visual-studio?view=vs-2019#test-the-code-and-compare-the-results>

[19] SHUBHAM, *Install OpenCV C++ with Visual Studio on Windows PC*. Decipher Technic, 8 mayo 2018. [online]. Disponible en: <https://www.deciphertechnic.com/install-opencv-with-visual-studio/>